Solving large stochastic planning problems using multiple dynamic abstractions

by

Kurt Alan Steinkraus

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

Author .... [illegible]
Department of Electrical Engineering and Computer Science
April 25, 2005

Certified by .... [illegible]
Leslie Pack Kaelbling
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chair, EECS Committee on Graduate Students

Solving large stochastic planning problems using multiple dynamic abstractions

by

Kurt Alan Steinkraus

Submitted to the Department of Electrical Engineering and Computer Science
on April 25, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

One of the goals of AI is to produce a computer system that can plan and act intelligently in the real world. It is difficult to do so, in part because real-world domains are very large. Existing research generally deals with the large domain size using a static representation and exploiting a single type of domain structure. This leads either to an inability to complete planning on larger domains or to poor solution quality because pertinent information is discarded.

This thesis creates a framework that encapsulates existing and new abstraction and approximation methods into modules and combines arbitrary modules into a hierarchy that allows for dynamic representation changes. The combination of different abstraction methods allows many qualitatively different types of structure in the domain to be exploited simultaneously. The ability to change the representation dynamically allows the framework to take advantage of how different domain subparts are relevant in different ways at different times. Since the current plan tracks the current representation, choosing to simplify (or omit) distant or improbable areas of the domain sacrifices little in the way of solution quality while making the planning problem considerably easier.

The module hierarchy approach leads to greater abstraction that is tailored to the domain and therefore need not give up hope of creating reasonable solutions. While there are no optimality guarantees, experimental results show that suitable module choices gain computational tractability at little cost to behavioral optimality and allow the module hierarchy to solve larger and more interesting domains than previously possible.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor of Computer Science and Engineering

Acknowledgments

I would first like to thank my thesis advisor, Leslie Pack Kaelbling. I am very grateful for the six years we have had to work together at MIT, and I have learned a lot not only about artificial intelligence but also about reading papers and distilling their essence quickly, writing clear and clean prose, and figuring out the important questions to ask. Leslie has supported me in discovering what I am interested in, never insisting on her own agenda, and she always has found time to discuss my latest results no matter how busy she might be. I am thankful for her broad artificial intelligence knowledge, infectious optimism, and thoughtfulness. I don't think I could have had a better thesis advisor.

Many thanks also to the other members of my thesis committee, Bruce Blumberg, Tom Dietterich, and Pete Szolovits. Their comments were very helpful in suggesting new areas of exploration, overcoming difficulties I was facing, and sharpening the presentation of my research.

Thanks to the members of my lab, including Bill, Selim, Jeremy, Terran, Kevin, Georgios, Hanna, Bruno, Leon, Mike, Yu-han, Natalia, Luke, Sarah, James, Meg, Sam, Ryan, and Teresa. I am thankful for the time we spent together comparing notes on artificial intelligence algorithms and for all the encouragement I received. Our statistical AI reading group was always a source of new and interesting ideas and thought-provoking discussion.

Thanks to John Laird and the SOAR group at the University of Michigan, where I completed my undergraduate degree. My time as a undergraduate research programmer there piqued my interest in artificial intelligence and provided me with understanding and experience that proved very valuable in my doctoral work.

Finally, I would like to thank my family. I am grateful to my parents, for their unconditional support and for instilling in me a love of learning from a very early age. I am especially grateful to my wife, Karen. She has been exceedingly patient and understanding during the long hours and late-night programming sessions that completing my research required, helping me keep perspective and not forget about living life. I am indebted to her for her kindness and love.

This thesis was supported in part by NASA award #NCC2-1237 and in part by DARPA contract #DABT63-99-1-0012.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the goals of AI is to produce a computer system that can act intelligently. Such a computer system would be able to gather data about the world, reason about how to achieve certain goals, and act so as to achieve those goals. The sorts of situations in which such a computer system would be useful include mobile robot and vehicle control, manufacturing plant operation, resource scheduling and distribution, and so on. These scenarios involve sensing the current conditions of the world, planning the best course of action given the stated goals, and ensuring the proper execution of that plan. There are many challenges to be met in producing such an intelligent computer system, at all stages of its operation.

In many cases, it is difficult to ascertain the current conditions of the world. Sometimes this is due to the current conditions being inherently unobservable, as would be true of the opponents' cards in a card game. Most often, though, this is due to the system being equipped with only a limited number of sensors or perhaps a type of sensor ill-equipped for the task at hand. That the world is only partially observable requires a computer system to deal with uncertainty by estimating (or at least compensating for) what it cannot see. Describing and modeling absolutely everything that a computer system cannot sense is often an intractable task, and approximations and heuristics must be used with care to ensure that vital information is not ignored or discarded.

Along with the ability to take sensor readings, it is often advantageous for a computer system to have a model of how the world changes, where the changes are due both to actions that the agent takes and to the dynamics of the world that are not agent-based. This model may be held implicitly in the way that the system converts stimuli into responses, or it may be explicitly stored and manipulated. A common way of representing the state and dynamics of the system's domain is as a Markov decision process (MDP) [36]. MDPs model domains where the current state of the world is always visible, and they allow for stochasticity and uncertainty in the domain's dynamics. MDPs are models that satisfy the Markov condition, meaning that the state of the world evolves based only on the immediate past, and not on the distant past. This property is desirable because it alleviates the need for the system to remember things that happened more than one time-step ago; all the relevant information about the world is encapsulated in the current state for MDPs.

Given an MDP model, is theoretically easy to figure out how to act so as to maximize expected reward. A solution to an MDP is a policy, which is a function that says what action to take given the domain's current conditions. Infinite-horizon discounted MDPs are guaranteed to have an optimal policy that is stationary, i.e., that can be written down once and doesn't change over time. There are common MDP-solving algorithms, such as value iteration and policy iteration, that start with an approximate solution and reach an optimal one through iteration. These algorithms perform the planning step, deriving the best possible course of action, and then there is no need to think any more. Since the domain's dynamics are known and accounted for in the calculated policy, the system can act optimally in the domain simply by looking up the best action for the current state in the policy table mapping states to actions.

A major drawback to representing domains as MDPs is that there will be one state for every possible configuration that the domain can be in, and likewise there will be one MDP action for every possible combination of domain actions that can be taken simultaneously. This has several consequences.

The first consequence is that most interesting domains have huge numbers of states and actions. An intuitive idea of the potential enormous size of state and action spaces comes from viewing them as being composed of independent entities that each can have one or more configurations or choices. For instance, if a mobile robot can be at ten locations, and its left and right gripper arms can have ten configurations each, then the total size of this state space is the product of the individual spaces, i.e., a thousand states. This exponential blowup of the size of the space also occurs with the action space. It is apparent that, for real world domains, the state and action spaces are not even tractably enumerable, let alone amenable to policy calculations. Any algorithm that relies on examining each state, such as value or policy iteration, is doomed to fail after the domain size increases past a few hundreds of thousands of states.

One approach to dealing with large Markov models has been to manipulate the model with the state space being factored into state variables (and likewise with action space); the whole state space then is the cross product of the possible state variable values, as discussed above. This allows the dynamics of the system to be specified in a similarly factored fashion, as in Bayesian networks [59]. If there is conditional independence structure in the domain, the factored dynamics allow for more efficient computation involving large uniform areas of state and action space, and therefore they potentially allow more efficient policy computation.

A second consequence of representing domains as MDPs is that reasoning will not necessarily be shared between similar states and actions. If a mobile robot is carrying a red package to deliver, then that will be a different state than if it were carrying a blue package to deliver, and it will not automatically be able to share information between the two situations about the best routes to take and so forth. There are other types of spatial and temporal structure present in domains but not made explicit in the standard MDP representation, it is crucial to recognize and take advantage of these similarities if huge domains are to be rendered tractable.

The factored MDP representation somewhat alleviates this problem, and other researchers have attempted to exploit structure in parts of the model description with explicit structural representations, like using decision diagrams to specify transition probabilities. (See the discussion in chapter 2 for examples.) Other researchers have tried integrating extra domain-specific information, where the extra information is specified by a human expert, derived from the model, or learned through trial runs. Yet other researchers have devised methods to speed up algorithms by approximating or distributing different steps; for instance, an area of state space that behaves uniformly (or approximately uniformly) may have its states' differences abstracted away and be operated on as one state.

Despite much effort, most very large domains remain intractable. One reason for this is that most existing algorithms to solve large MDPs take just one simplification step in attacking the problem, using one kind of abstraction or approximation. To see why this will not work, imagine trying to solve a huge domain by subdividing the task into two subproblems: creating an abstraction, and solving the abstraction. If the abstraction is very close in content to the original domain, then it is likely to be just as huge and just as intractable. On the other hand, the abstraction may be relatively small and tractable, perhaps because the algorithm that created it has mandated a maximum size for it. Such an approximation of the original domain is only tractable because all but a small pre-determined amount of information has been discarded. Clearly, unless the algorithm is enormously clever in deciding what information to discard, the policy obtained from solving the abstraction will be unable to navigate the complexities and nuances of the original domain and is likely to fail to achieve good performance.

A second downside of taking just one simplification step is that only one kind of structure is exploited, and this does not match well with the varying kinds of complexity that most large real-world domains exhibit. Granted, some very large problems are inherently difficult and cannot be approximated or distributed to make them easier. An example might be a stochastic traveling salesman problem with some large number of cities, where the problem is known to be NP-complete. It is unsurprising that these domains cannot be tractably solved optimally. Other large real-world problems are highly structured, perhaps containing many repeated slight variations of some simple task. The extremely structured nature of such problems renders them solvable despite their size. The vast majority of interesting problems, though, lie in between these two extremes: they contain a low proportion of inherent difficulty, yet they are not so structured that they are embarrassingly easy to solve.

Most existing algorithms not only use just one simplification step, but they also use just one domain representation, where the domain representation here includes not only the input but also any additional abstraction and approximation data that are calculated in the process of planning and executing. This domain representation is usually calculated during the planning phase, before execution has begun, and once calculated, it is never revisited. Indeed, were it to be revisited, it would not change at all, since most methods operate on the whole domain at once and have no notion of, e.g., the current relevancy (or lack thereof) of part of it. Calculating a policy for the whole domain all up front means that the system is required to plan for the whole domain all at once, and even with the more compact representations like factored MDPs, this can be an intractably large task.

This thesis addresses solving very large factored MDPs by alleviating the problems of one simplification step and one domain representation. It proposes a dynamic system called a *module hierarchy* that combines multiple different abstraction algorithms and changes the domain representation during execution to suit the current state. The module hierarchy packages abstraction methods into modules that share a common interface, making them easily combinable in whatever way best suits the domain to be solved. During execution, the module hierarchy maintains a series of progressively more abstract models of the domain and cascades opportune dynamic representation changes from one model to the next.

The module hierarchy framework provides a principled way of successively abstracting and approximating a domain until an easily solvable core remains, then planning and executing using the exploited structure. It does not guarantee hard optimality in the policy that it derives (except in very special cases), but this is to be expected since no tractable algorithm guarantees hard optimality. The module hierarchy instead aims for tractability while still acting in a reasonably good manner. It focuses on achieving this for domains that are arranged as the real world is: containing an array of different types of exploitable structure.

Before surveying existing abstraction methods in more detail and then seeing how the module hierarchy framework can improve on them in dealing with very large models, it is necessary to define MDPs of different kinds more precisely and to talk about their advantages and disadvantages.

1.1 Markov decision processes

A Markov decision process (MDP) [36] is a Markov model with actions. An MDP is composed of several parts:

- A set of states $S = \{s_1, s_2, \ldots, s_n\}$, where S is the possible configurations of the world. The world will always be in some state in S.

- A set of actions $A = \{a_1, a_2, \ldots, a_m\}$, where A is the possible things that an agent can do at any time-step. The allowable action set could potentially vary from state to state, but, without loss of generality, assume that it is constant; if an action is inapplicable in some state, then perhaps it performs a no-op.

- A transition probability distribution $T : S \times A \times S \to \mathbb{R}$, which gives a probability distribution over possible next states that is conditioned on the current state and the currently chosen action. Note that, for all $s \in S$ and $a \in A$, $\sum_{s' \in S} T(s, a, s') = 1$.

- A reward function $R : S \times A \rightarrow \mathbb{R}$, which gives the reward that the agent receives conditioned on the current state and the agent's currently selected action.
- A starting state probability distribution $start : S \rightarrow \mathbb{R}$, which gives the possible states that the agent can start at along with their probabilities.
- A discount factor γ, where $0 < \gamma \leq 1$.

The MDP is a *decision process* in that it represents a process in the world in which an agent participates by making sequential decisions. A *Markov* decision process is one where the Markov property holds: the probability distribution of states at the next time-step is only dependent on the current state and the agent's currently selected action; in particular, the probability distribution over next states is conditionally independent of all past states and actions given the current state and action. More formally, given a sequence $s_1, a_1, s_2, a_2, \ldots, s_i, a_i$ of alternating states and actions,

$$p(s_{i+1} = s^* | s_1, a_1, s_2, a_2, \ldots, s_i, a_i) = p(s_{i+1} = s^* | s_i, a_i).$$

There are many different classes of MDPs; this thesis focuses on completely observable infinite-horizon discounted finite state and action space discrete time MDPs. In particular:

- The current state is assumed to be completely observable. This is a big assumption and does not hold for many real-world domains, but the complexities that this thesis attempts to unravel are orthogonal to the observable/partially-observable distinction.
- The agent gets an overall reward for acting in the domain, and the overall reward is generated from the individual instantaneous rewards that the agent

receives by summation, discounting future reward:

$$\text{Total reward} = \sum_{i=1}^{\infty} \gamma^{i-1} R(s_i, a_i)$$

Other common formulations of the reward term to be maximized are the finite and infinite horizon average reward:

$$\text{Finite horizon average reward} = \sum_{i=1}^{n} R(s_i, a_i)$$

$$\text{Infinite horizon average reward} = \lim_{n \to \infty} \sum_{i=1}^{n} R(s_i, a_i)$$

This work applies to both of these total reward formulations with the obvious caveat that the abstraction modules used in the abstraction hierarchy must be compatible with whichever total reward formulation is chosen.

- The state and action spaces are finite (and therefore discrete). It would be useful to be able to act on continuous or hybrid state and action spaces, and there is no clear reason why this work should not extend to these cases.

- Time advances in discrete unit steps rather than continuously. Certain simulations involving fluid dynamics or similar phenomena are described using continuous partial differential equations and are very difficult to solve. The sorts of very large, complex, and heterogenously structured domains that a mobile robot or software agent are likely to act in, though, are either inherently discrete or can be normally be discretized without losing the essential domain qualities.

Given a domain expressed as an MDP, the goal is to act in that domain so as to maximize the discounted long-term expected reward. Solving a deterministic domain involves creating a *plan*, a step-by-step list of what to do at each step. MDPs, however, cannot be solved by creating a plan, because the stochastic nature of actions almost certainly means that actions need to be selected in an on-line fashion, i.e., based on the results of previous actions. Solving an MDP involves creating a *policy*, a mapping from states to actions telling what to do at each state. Plans or policies may be represented as tables or trees or any other data structure, and they may also be constructed in a just-in-time manner by some algorithm; the only important thing is that they dictate what action to take next.

1.2 MDP structure

An MDP as defined in the previous section has components (a state space, an action space, a transition probability function, a reward function) that have no specified structure. Thus, the state and action spaces could be represented by simple lists of states and actions, and the transition probability and reward functions could be stored as big tables.

Storing everything in this flat way is frequently undesirable. It is time- and space-consuming to store even medium-sized MDPs in this way; very large MDPs are completely intractable. Most domains have some sort of structure, though, and if an MDP is being specified by a human or is being learned by an agent, then whatever structure the human or agent knows of should be captured in the domain description. As much of this structural meta-data as possible should be retained, because it is easy to throw away structure if it is not useful, but it can be difficult to discover it from unstructured data.

There are several common types of structure used in specifying MDPs: structured state/action spaces, structured functions, and temporal extension.

Factored MDPs One type of structure involves the state and action spaces being factored into variables, so that the range of values for the whole space is the cross-product of the variables' values. This is similar to a dynamic Bayesian network (DBN) [15], except that there are also actions instead of just states. The transition probability and reward functions are defined over state and action variables. The transition probability function can be factored just like the state space, so that there are several component transition probability functions, one for each state variable, and the probability of making the transition from one state to another is given by the product of the component transition probability functions. Since it will not be the case that all state variables affect the next time-step value of all state variables, each component transition probability function will generally restrict its domain to a small number of state variables and will be explicitly independent of the rest. This explicit conditional independence is what allows for the majority of the space savings in specifying an MDP in a factored way.

Formally, a *factored MDP* is composed of several parts:

- A set of state variables $\overline{S} = \{S_1, \ldots, S_N\}$ where each S_i is a state variable taking values $\{s_{i1}, \ldots, s_{in_i}\}$; the state space as a whole is given by $S_1 \times \ldots \times S_N$.

- A set of action variables $\overline{A} = \{A_1, \ldots, A_M\}$ where each A_i is an action variable taking values $\{a_{i1}, \ldots, a_{im_i}\}$; the action space as a whole is given by $A_1 \times \ldots \times A_M$.

- A set of component transition probability functions $T = \{t_1, \ldots, t_N\}$ where each t_i is a conditional probability function that gives the probability of states at time $\tau + 1$ given the state and action at time τ; the range of state and action variables for t_i are given by $\overline{S_i} \subseteq \overline{S}$ and $\overline{A_i} \subseteq \overline{A}$ respectively, and

$$t_i : \prod_{S \in \overline{S_i}} S \times \prod_{A \in \overline{A_i}} A \times S_i \rightarrow \mathbb{R};$$

  since t_i is a conditional probability distribution, for any state s and action a, $\sum_{s' \in S_i} t_i(s, a, s') = 1$.

- A reward function r where $r : \prod_{S \in \overline{S}} S \times \prod_{A \in \overline{A}} A \rightarrow \mathbb{R}$.

- A discount factor γ, where $0 < \gamma \leq 1$.

Structured component transition probability functions Factored MDPs allow for some structure of the state and action spaces and the dynamics of an MDP to be exposed, but each component transition probability function might still be written down in table format; if the component transition probability function is conditioned on a large number of state variables, then this table will necessarily be very large. The same is true for the reward function.

Researchers have proposed structuring component transition probability and reward functions in different ways. One promising way is as decision trees [8], where each choice point in the tree is labeled with a state variable, each branch is labeled with a possible value for that state variable, and each leaf has a real value giving a probability or a reward. The benefit of a decision tree is that it can compactly represent identical values or identical patterns of values for whole areas of state and/or action space.

Algebraic decision diagrams [34] build on decision trees. Decision diagrams are like decision trees except that a node may have more than one parent; in other words, two routes from root to leaf may branch and then join back up further down the diagram. In a decision diagram, no subpart may be repeated; if a subpart would appear in two places, then those copies are merged, and all parents point to the merged version.

Decision diagrams carry the advantage over decision trees of possibly being more compact and of there being one canonical decision diagram for any given function to represent and any ordering of choices from root to leaves. However, decision diagrams are also more computationally expensive to maintain due to the no-repeated-subparts requirement.

Semi-MDPs MDPs as described so far have actions that take exactly one time step. It is often advantageous, though, to be able to represent that different actions take different lengths of time to complete, and indeed, a single action may take probabilistically varying lengths of time to complete. A semi-MDP is an extension of an MDP that allows actions to take differing amounts of execution time.

Formally, a semi-MDP is composed of the same things as a normal MDP (state space, action space, transition function, reward function), except that the transition function T gives transition probabilities not only for the new state but also for the time that the transition will take:

$$T : S \times A \times S \times \mathbb{N} \rightarrow \mathbb{R},$$

and

$$\text{For all } s \in S \text{ and } a \in A, \sum_{s' \in S, t \in \mathbb{N}} T(s, a, s', t) = 1.$$

Other types of MDP structure There are many additional types of MDP structure, involving techniques like logic [62] and first-order representations [9]. The module hierarchy described here does not operate on these other types of MDP (or rather, it doesn't exploit their special nature; of course it could always operate on them after grounding all terminals and flattening the state and action spaces). See section 7.2 for discussion of how it might be extended to these additional representations.

1.3 Why MDPs?

The question arises, why use a Markov decision process to model the world and to plan in?

One possible alternative would be to model the world using a deterministic rather than a stochastic model. In the artificial intelligence community, planning started out in deterministic worlds, and indeed this made planning considerably easier. In deterministic worlds, it is possible to come up with a straight-line plan, as opposed to a policy, because there will never be an unanticipated event. In deterministic worlds, it is much easier to look ahead because there is not the branching factor at each time step that gives rise to exponential growth in the search tree size.

Despite their advantages, deterministic models are lacking in other areas. The first and most obvious is that deterministic models cannot model uncertainty or stochasticity in the world. It is an interesting philosophical question to ponder whether saying, "the probability of event X is one in two" means that event X is a probabilistic event or if it means that not enough is known about the current state of the world to be able to determine whether event X will occur, even though the process may be deterministic. Luckily, stochastic models do not force committal to a specific metaphysical interpretation of probability and can be used to model either uncertainty about the world or true stochasticity in the world.

(Using stochastic models for modeling uncertainty about the world is one reason that assuming completely observability in an MDP is not always an unrealistically simplifying assumption. The partial observability in the world can always be modeled as stochasticity, but then the model may no longer satisfy the Markov condition, and it is not always the case that this is a helpful way to model the domain.)

It will also turn out that deterministic models are lacking because they cannot represent certain types of approximations very well. For instance, suppose an agent is building a large plan, and part of its plan will involve the completion of some subtask. If the agent knows that the subtask can be completed, then it can defer the step-by-step planning until later. A deterministic model will need to estimate a single time-length for how long the subtask will take to complete; a stochastic model, on the other hand, can use a bimodal distribution, or any sort of distribution it wants. As another example, an agent may want to aggregate states together, giving the aggregate state probabilistic dynamics according to the outcomes from each included state (see section 5.1.2); this would not be possible in a deterministic model. Certain models and planning strategies such as conformant planning [77, 5] work with models and assumptions that lie somewhere between full deterministic and full stochastic models and show promise of combining the advantages of both.

One limitation of using an MDP is its reward structure: it assumes that the rewards are gained independently, that they sum, and that they are discounted over time. Some work is currently being done with non-additive rewards (e.g., [20]), and this limitation can also be worked around since a stochastic domain with non-additive rewards can be turned into a discounted MDP by artificially augmenting the state with some or all of the reward history.

Another limitation of using an MDP to plan in is that it is necessary to have the full-blown model before planning occurs, and the model is assumed to be correct, i.e., to reflect the dynamics of the domain being modeled accurately. For some domains, it is simply tedious to write down the whole model; for others, the dynamics may even be unknown. The module hierarchy described here uses MDPs because in order to concentrate on the planning aspects (rather than learning aspects) of working with stochastic models of very large domains. Other work has been done on learning Markov models of a domain (e.g., [41]). Later sections contain details on how this learning can occur concurrently with planning and execution, and how the system can efficiently incorporate newly learned dynamics into the current execution path.

Chapter 2

Previous work

There are very many approaches to planning and acting in very large MDPs. These methods try to take advantage of the structure of the problem to help find good solutions. Sometimes the structure is gleaned from the model itself, and sometimes a description of the structure is supplied on the side. In either case, this extra information is normally used to compress or abstract away details of the domain, to make it more tractable.

Methods can be divided into several categories based on the main method that they use to attack the problem. These categories are based on algorithm characteristics like the particular input data they operate on, the goal of the abstraction, etc. This section discusses baseline methods for solving MDPs and then outlines a variety of abstraction and approximation algorithms. While these algorithms may not be able to solve extremely large planning problems on their own, they can still be viewed as candidate components for a larger system.

2.1 Baseline MDP methods

There are several methods for directly finding optimal policies for MDPs. These methods are have simple algorithms and are therefore easy to implement, but their running time is quadratic or worse in the size of the state and action spaces. This makes them suitable for doing baseline comparisons and for checking the correctness of other methods on small domains, but they're not fast enough for real-world use in very large domains.

2.1.1 Value and policy iteration

The simplest way of finding an optimal policy for an MDP is called value iteration [36]. Letting $v^*(s)$ be the optimal expected value of being at state $s \in S$, the Bellman equation [3] describing the optimal value function is

$$v^*(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') v^*(s') \right] \quad \text{for all } s \in S.$$

The algorithm starts with some initial value function, say, $v_0(s) = 0$ for all $s \in S$, and uses an iterative form of the Bellman equation,

$$v_{i+1}(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') v_i(s') \right] \quad \text{for all } s \in S.$$

The sequence of value functions v_i is guaranteed to converge to the optimal value function v^*, and then the optimal policy π^* is obtained by following a greedy one-step lookahead policy,

$$\pi^*(s) = \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') v^*(s') \right].$$

Policy iteration for MDPs, like value iteration, involves iteration and use of the Bellman equation to find the best action at states. However, policy iteration continually updates a policy π rather than a value function over successive iterations. The two steps it alternates between are:

1. Calculate v_{π_i}, the value function for policy π_i, where

$$v_{\pi_i}(s) = R(s, \pi_i(s)) + \gamma \sum_{s' \in S} T(s, \pi_i(s), s') v_{\pi_i}(s').$$

2. Update π by calculating

$$\pi_{i+1}(s) = \arg\max_{a \in A} \left[R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') v_{\pi_i}(s') \right].$$

In the first step, the value function is calculated in much the same way as the value iteration algorithm, except that the absence of a maximization over actions means that this value function calculation converges much more quickly. (The value function can also be calculated using matrix inversion.) In general, policy iteration converges more quickly than value iteration.

Puterman and Shin noted that the first step doesn't need to produce a very precise value function, but rather it just needs to get close enough for the policy update step to be valid. This led them to create the modified policy iteration algorithm [67, 66], which is just like policy iteration except that the value function calculations are stopped early before they converge, according to some criterion which guarantees that the overall policy iteration still converges to the correct value.

Aside from being used iteratively, the Bellman equation can be expressed as a linear program and solved using a linear program solver.

2.2 Prior abstraction methods

2.2.1 Pure structure exploitation approaches

Some of the prior work has not so much focused on coming up with new algorithms to solve MDPs as it has coming up with ways to represent MDPs and to exploit the explicit structure in the representation while executing existing algorithms like value and policy iteration.

Explanation-based reinforcement learning Explanation-based reinforcement learning (EBRL) was created by Dietterich and Flann [19] in an attempt to combine explanation-based learning (EBL) and reinforcement learning (RL). EBL allows an agent to learn how to act in large swaths of state space at once, by working backward from gained reward and figuring out where to assign credit (and what to ignore). It can do this working backward efficiently since the domain is represented in a probabilistic STRIPS format. RL, on the other hand, allows an agent to gradually form a policy, learning from experience and correcting earlier mistakes in policy construction. EBRL combines these two.

The EBRL algorithm consists of several trials. In each trial, a random starting state is selected, and the agent follows the current best policy until it reaches the goal. Once the goal is reached, a series of value function backups is performed from the goal back to the starting state. These value function backups are region-based, i.e., they are like normal point-based RL Bellman backups, except that only a certain path is followed (instead of the whole state space being backed up), and whole regions of state space are backed up simultaneously. This region backing up works because the value function is stored as a collection of hyper-rectangles in state space, each with the associated value. When backing up some regions, certain rectangles in the value function may need to be split.

EBRL is essentially EBL that updates a value function at each time-step rather than a policy. Mixing in RL's value function in this way allows it to avoid being sensitive to (and possibly misled by) the path it took the first time it visited each state, which is a common flaw of standard EBL algorithms. If just a policy were to be constructed, there would be no good way to resolve conflicts about what action to take in certain regions; by using a value function, the policy is only stored implicitly, and the policy will change as needed as that value function is updated.

Structured policy iteration Boutilier *et al.* developed a variant of modified policy iteration called structured policy iteration (SPI) [8]. The structured policy iteration algorithm maintains a current best policy and a current best value function, both represented as decision trees. Beginning with some randomly selected initial policy, the algorithm alternates between the policy iteration steps of (a) updating the current best value function by estimating the value function associated with the current best policy, and (b) updating the current best policy to be equal to the the best one-step lookahead policy for the current best value function.

The first update step, which estimates the value function for a given policy, is called *structured successive approximation.* This step performs structured value iteration, starting with some initial guess at the value function (the instantaneous reward function) and successively refining its estimate until the difference between successive iterations is below some threshold. Each Bellman backup is performed on the tree-structured value function in a structured way, which is possible because two states with the same current value will have different values in the next iteration only if some action taken at those two states has differing effects and those differing effects are relevant to the current value function. This allows the Bellman backup to be done for each leaf of the tree rather than each state in state space.

The second update step, which builds a new policy from a value function, is called *structured policy improvement.* This step incrementally improves the current best policy to create a new best policy. It uses the same sort of reasoning as the first step to perform this incremental policy improvement in a structured way, on leaves of the policy tree rather than at every state in state space.

SPI takes advantage of not only factored MDP structure but also tree structure in the transition and reward probability functions. It thus improves on modified policy iteration by allowing it to exploit the structure exposed in the domain description.

Factored value functions The factored value function method of Koller and Parr [42, 43] takes advantage of the structure in factored MDPs to represent the value function in a similarly factored way. It combines policy iteration with its factored value function representation by projecting back into the factored value function space after each step of policy iteration.

This method requires not only that the domain be a factored MDP but also that it have fairly uniform action dynamics: most state variables have some default dynamics for most actions and non-default dynamics for only a few actions. This method has a special factored representation for value functions and for policies. It represents the former as a linear combination of basis functions, where each basis function takes as arguments only on a small number of state variables. A policy is represented as a decision list consisting of tuples, where each tuple contains a partial assignment to state variables and an action to take if the partial assignment matches the current state.

The size of a policy thus naturally depends on the domain complexity, as manifested by the type of factored structure present in the MDP, rather than on the state and action space sizes. Also, representing only policies whose value function lies in the factored value function space is good for efficiency, both because it is a smaller space to search than the whole of policy space, and because it allows structured policy iteration operations. Koller and Parr give the steps for doing policy iteration using these factored representations so that it is not necessary to enumerate state space explicitly.

SPUDD and APRICODD Structured policy iteration using decision diagrams (SPUDD) [34, 35] and approximate policy construction using decision diagrams (APRICODD) [78] are extensions of structured policy iteration. Instead of representing the domain's transition and reward functions as decision trees, though, these algorithms represent them using algebraic decision diagrams (ADDs).

(Recall that decision diagrams are like decision trees, except that branches can and do merge back together. Given a particular function to represent and a particular ordering of variables from top to bottom in the decision diagram, there is exactly one corresponding ADD that represents the function.)

SPUDD is a straightforward extension of SPI; it takes advantage of the extra structure in ADDs, and by using ADDs rather than decision diagrams, the representation is potentially orders of magnitude smaller.

APRICODD takes SPUDD a step further by introducing approximations. Each function represented as an ADD has, at its leaves, not a single value but rather a range of values. At each iteration in policy iteration, APRICODD tries to merge ADD leaves while making sure that the size of the range of values that each leaf represents remains below some bound. Using these approximate ADD functions makes policy iteration take fewer iterations to converge and dramatically reduces the size of the ADDs.

Logical Q-learning Kersting and De Raedt added a different type of structure to MDPs and created logical Markov decision programs (LOMDPs) [40]. In an LOMDP, the states and actions are given by logical formulas that are conjunctions of facts about the domain, and the transition and reward functions are given by logical probabilistic STRIPS-like rules.

Kersting and De Raedt demonstrate how to create a policy for a LOMDP using a Q-learning variant called LQ-learning. The policy is a list of state-action rules and is executed by taking the first action whose corresponding state matches the current world state. These states in the policy list are abstract, in the sense that they are composed of conjunctions of facts about the domain, and these abstract states are said to match any fully ground state given by a conjunction of facts that extends it.

Given a set of abstract states that covers state space, LQ-learning performs Q-learning to determine the correct action to take at each abstract state. LQ-learning forms an abstract policy that works well for the domain in which it was trained;

however, since the policy is abstract, it will (ideally) work well in any domain that has the same predicates as the training domain but different constants.

RMDP plan generalization The algorithm for plan generalization in relational MDPs of Guestrin *et al.* [26] adds yet another type of structure to MDPs. In a relational MDP (RMDP), the domain is defined using a schema, much like a relational database. The schema specifies classes of objects, and each class has state and action variables associated with it. The schema also specifies the types and parameters of possible relations between objects, and it gives general transition and reward dynamics parameterized by object relations. A particular domain is an instantiation of a schema, containing instances of objects and instances of relations between those objects.

Given one or more RMDP domains, the RMDP plan generalization method learns to act in the domain in such a way that the information it learns will easily generalize to other RMDPs with the same schema. The method approximates the value function for the whole domain as the sum of the value functions for the objects in the domain. It then assumes that objects of the same type will contribute to the overall reward in the same way, so their component value functions do not need to be learned separately.

To create a plan, the method first operates on some small example domains with the same schema. It uses linear programming to find the overall value function formulated as a sum of component value functions for objects. It then applies this value function to one or more large domains simply by adding or removing component value functions based on the specific objects in the large domains. Guestrin *et al.* show that the value function is close to the optimal one in that class (sum of component value functions) with high probability given a polynomial number of worlds to learn from.

2.2.2 Divide and conquer approaches

The second general technique for solving MDPs is divide-and-conquer. Methods falling into this category work very well when the domain can be taken apart into pieces that only weakly interact. There are many different ways that the domain can be subdivided (over the state space, over state variables, over the action space, etc.), and these methods run the gamut. The difficult part in all of these algorithms comes when it is time to join the separately solved parts; the joining is always smoother or quicker if the initial partition didn't divide up strongly interacting subparts.

Hierarchical policy construction Dean and Lin propose several decomposition techniques [16]. The algorithm takes as input a factored MDP and a division of that MDP's state space into regions where only a small number of state variables are relevant to any subpart.

The algorithm alternates between local and global computations. In the first step, it creates a set of policies for each region separately, where policies are created for various costs associated with making a transition out of that region. The set of policies ends up containing all policies necessary to get from one boundary state to another, given some "urgency" cost. In the second step, after solving each region by itself, the algorithm constructs an abstract MDP whose abstract states are the regions and whose abstract actions are the policies calculated to go from region to region. This abstract MDP is solved, and then a thresholding process determines whether to continue iterating (this time using different costs for region transitions) or whether the current overall policy is close enough to optimality.

Prioritized goal decomposition Boutilier *et al.* [6] propose an algorithm, called prioritized goal decomposition, that breaks an MDP's reward function into summands. For each additive component of the reward function, a plan is found that optimizes that reward component, and then the algorithm merges these component plans into one. (The reward function is assumed to be additive with terms referring to different state variables.)

The idea behind the algorithm is to create least commitment plans (LCPs) for maximizing each reward function component. These LCPs are not full-blown policies but rather are plan pieces; they are composed of a set of actions, a set of causal links dictating what actions produce the preconditions required by other actions, and a set of ordering constraints on the actions. Each LCP can be *linearized* to form a policy that optimizes a particular reward function component, but each LCP is more general than any specific policy.

Once LCPs have been created for each reward function component, the LCPs are merged. This merging works by using the sum of the value functions for each component as an admissible heuristic for A*. To make the A* search tractable, the merging is done pairwise starting with the policy of potentially highest value, and at each merge the merged policy is constrained to obey the causal structure of the LCP for the prior merged policy. This commitment to the LCP for components already merged may cause the resulting policy to be non-optimal, but it causes a big reduction in computation time.

Cached policy sets The cached policy sets algorithm by Parr [56] divides up an MDP's state space into regions and creates a cache of policies for each region. Parr notes that state space can often be divided into subparts that are weakly connected, i.e., there are few states where actions can be taken to transition from one subpart to another.

In contrast to previous methods that pass messages back and forth between regions and continually update each region's policy, a set of policies is created for each region, where the set is guaranteed to contain at least one policy that performs within a constant of optimal regardless of the structure of other regions. The insight that allows this is that the value at each state in the region is a linear function of the regional policy and of the long-term expected values of states outside the region, assuming the latter two are fixed. The caches are therefore built iteratively by continually adding policies that handle the values of states outside the region for which the policy set as a whole performs worst.

The cached policy sets algorithm creates an abstract semi-MDP whose abstract states are the regions' border states. The abstract actions needed to solve the abstract semi-MDP (i.e., the ones in its optimal policy) are mapped into goals for the regions, and the regions build policy caches as discussed above. Finally, the regions' caches of policies are combined to create dynamics for the abstract semi-MDP actions, and a global policy is calculated that performs within some constant of optimal.

Parr notes that the policy caches for some regions may need to be very large if they are to provide policies within some factor of optimal no matter how the other regions are structured. He thus gives a variant of this algorithm that builds a small cache for each region and then calculates (and caches) a new policy when it is deemed advantageous enough to do so rather than to use one of the existing cached policies.

Dynamic MDP merging The dynamic MDP merging method of Singh and Cohn [76] attacks the problem of what to do if you have multiple MDPs to act in simultaneously, the total reward is the sum of each MDP's reward, and choosing an action in one MDP constrains the actions you can take in another (e.g., you have a limited number of resources to spend at each time-step).

The algorithm starts with a set of MDPs and, for each MDP, bounds on the value function at each state. The algorithm performs a variant of value iteration in the combined space of all the MDPs. Instead of updating the value for each combined state, though, the algorithm computes upper and lower bounds on the value at each state.

The algorithm avoids updating the combined value function for the entire combined state and action spaces in two ways. First, for each combined state, it maintains a list of combined actions that have been discovered to be suboptimal and need no longer be considered. Second, it updates combined states only along trial runs, where it starts at a start state and wanders randomly until it reaches a goal. In this way, the dynamic MDP merging is able to converge on the optimal policy much more quickly and efficiently than normal value iteration in the combined MDP space.

Hierarchical SMART The hierarchical SMART algorithm of Wang and Mahadevan [83, 84] performs hierarchical Q-learning for SMDPs. The inputs are several SMDPs that are loosely coupled, and the policy chosen in one SMDP may affect the transition dynamics or reward function at states in other SMDPs. For example, each SMDP may represent a machine in a factory, and the SMDPs are coupled by the output of one machine being fed into the input of another.

Flat Q-learning for the combined space of the SMDPs is intractable if there are many SMDPs. Hierarchical Q-learning is also relatively unsuccessful when learning Q values for each SMDP as it operates interacting with the other SMDPs. Hierarchical SMART, on the other hand, derives Q values for each SMDP in isolation, and it in fact calculates Q functions for many different values of the coupling variables. This creates a set of Q functions and therefore policies for each SMDP.

Once the SMDP policy sets have been created, an abstract SMDP can be build whose abstract states are the states of the coupling variables and whose abstract actions choose which policy to execute in each SMDP. Hierarchical SMART then does the second level of Q-learning, this time for the abstract SMDP, and combining the Q function at the abstract level with the policies in each SMDP gives a policy for the overall set of SMDPs.

The benefit of this algorithm as opposed to some previous ones that took similar approaches is that it does not rely on the coupling between SMDPs being due to a shared resource, or shared boundary states, or shared action constraints. Rather, it will work for any general coupling between the SMDPs.

Markov task decomposition The Markov task decomposition of Meuleau *et al.* [51] solves an MDP that is decomposed into a number of concurrent MDPs, where the total reward is the sum of each subpart's reward, and where subparts' actions constrain the allowable actions in other subparts.

The setup is very similar to that of dynamic MDP merging (section 2.2.2) except that Meuleau *et al.* concentrate on the case where there are so many MDPs that the explicit state and action space enumeration of dynamic MDP merging is intractable. Markov task decomposition avoids enumerating the spaces and gains tractability at the cost of guaranteed optimality.

The algorithm first considers each subpart separately, calculating a value function for each combination of start state, horizon length, and constraints on the subpart's actions. Once these individual value functions are calculated, the algorithm does not compute a global policy but rather chooses an action and then executes it at each time-step. When choosing an action for each subpart, the algorithm takes a two-step approach: it uses a domain-specific heuristic to assign resources to each subpart (i.e., to constrain the allowable actions in each subpart), and then it uses each subpart's pre-calculated value functions to figure out what action to take at that time-step.

Markov task decomposition may end up executing suboptimal actions even if the domain-specific heuristic optimally assigns constraints to each subtask, because the subtasks do not consider the probabilistic outcomes of other subtasks. Nevertheless, it tractably solves and provides a reasonably good policy for large decomposable domains where other algorithms cannot.

Castañon [10] and Yost and Washburn [85] solve domains in a similar way to the Markov task decomposition described above, except that they operate on POMDPs, not MDPs. Their methods use linear program to constrain the action for each subpart and to improve the resource constraints given best policies for each subpart.

Distributed hierarchical planning A somewhat different method is the distributed hierarchical planning done by Guestrin and Gordon [25]. The planning is done in a factored MDP by dividing it into subsystems; in particular, the state variables, not the state space, are divided into subsystems. Each subsystem is a set of strongly interacting state variables; subsystems may share variables and are arranged into a tree hierarchy according to the variables they share. (This algorithm is actually an extension of the factorized planning algorithm of Guestrin *et al.* [27], where both the planning and the execution are now distributed instead of just the execution.)

Each subsystem is composed of internal and external variables, with the intuition that the subsystem is in control of its internal variables and depends on but does not control the values of it external variables. The subsystems are solved individually but simultaneously and in interaction with other subsystems, as follows.

The MDP is represented as a linear program and is transformed into a compact approximate representation, one where the value function is assumed to be a linear combination of basis functions. The basis functions are chosen so as to allow complete flexibility in representing each subsystem's value function (over the subsystem's internal variables) separately.

The transformed linear program is then decomposed using a variant of Dantzig-Wolfe decomposition. Each subsystem has its own linear program that depend on other subsystems; in particular, each linear program has a set of inputs that can be interpreted as a reward adjustment. The subsystems participate in a message-passing phase, separately solving their linear programs and passing reward adjustment messages. These messages can be interpreted as a kind of reward sharing: each subsystem notifies its neighbors about how the neighbors' policies affect its own reward. When the message-passing phase converges, the resulting linear programs are combined into a global policy.

2.2.3 State aggregation approaches

A third category of abstraction methods has to do with state aggregation. The idea is to reduce the state space by grouping together certain states, either regularly or irregularly, according to some criteria. The algorithms that operate on factored input domains often cluster states by ignoring the values of certain state variables; this clusters together states that are the same except for their values over those ignored variables.

In general, the state abstractions are good because they don't require any extra input beyond the input model (though that input model may have to be specified in a special format, e.g., STRIPS notation for state space approximation). The state abstractions don't do any temporal abstraction, however, which limits the sorts of domain structure that they can take advantage of.

UTree The UTree algorithm of McCallum [47] (and the continuous state-space variation of Uther and Veloso [81]) progressively refine a tree-based aggregation of states by considering the state transitions encountered on trial trajectories and ensuring that the individual states inside aggregates have similar dynamics.

UTree is slightly different than most state aggregation approaches in that it doesn't simply partition the state space or the state variables. Instead, it assigns the current state to an aggregate based not only on the current state but also possibly on the recent history of states and actions as well.

The algorithm starts by grouping all states into one aggregate, and it then alternates between data gathering and refinement. In the data gathering phase, the agent uses the current aggregation to build an abstract model, and it performs Q-learning on the model, acting greedily. During this phase, it records the state, action, post-state, and reward for transitions that it makes. In the refinement phase, the recorded transition dynamics are used to calculate the Q values for each abstract state. If the Q values vary significantly over a state, it is split in two, where there are several possible criteria used to determine significant variance and to determine which state should be split.

The TTree algorithm of Uther and Veloso [82] is similar to their extension of UTree to continuous state spaces, except that it exploits user-supplied temporally extended actions. These sub-policies are used to guide the data gathering phase in exploring areas of state space that are more likely to occur on good trajectories.

Multigrid value iteration The multigrid value iteration algorithm of Heckendorn and Anderson [32] applies the multigrid approach to reinforcement learning in MDPs. Multigrid algorithms generally work by creating solutions for successively finer representations of a problem, starting with a very coarse abstraction or discretization, and ending when the representation is close enough to the initial problem to guarantee some sort of optimality.

Heckendorn and Anderson show how to do value iteration while varying the discretization from coarse to fine, stepping to the next finer discretization when value iteration almost stabilizes. They demonstrate that this algorithm converges faster in this varying discretization than value iteration does in a static fine discretization, and that the result is within a constant factor of optimal.

Model minimization The model minimization method of Dean and Givan [12, 13] creates a new model that is equivalent to the input model but that attempts to ignore all meaningless differences in state dynamics. It has two variants: one only ignores differences that are guaranteed not to make a difference, and the other approximates by additionally ignoring differences among states that are almost the same.

The algorithm constructs the abstract aggregations of states not by starting with all states separate and then grouping some together but rather by starting with all states grouped together into the same aggregate and then successively separating clusters until they meet a stability criterion. This stability criterion states that, given two aggregates S and S', and given an action a, every state $s \in S$ has the same transition probability to S' (i.e., to some any in S') under action a, and every $s \in S$ also gains the same reward under a. This refinement algorithm produces the coarsest refinement (i.e., with the fewest number of states) possible for which the abstract dynamics match the original dynamics.

The partition that the exact algorithm comes up with is often complicated and space-consuming to represent, since the partitions can have arbitrary shapes. Model minimization can be turned into an approximate but more tractable algorithm by choosing to limit allowed clusters to those that are easily or compactly representable, say, by conjunctions of literals. Dean and Givan show that model minimization is a generalization of state-space approximation (section 2.2.3) and of structured policy iteration (section 2.2.1), where these other spatial approximation methods change the refinement step to yield partitions with nice descriptions or partitions that are easier to find, rather than the fewest strictly optimal partitions possible.

Ravindran and Barto [68, 70] and Givan *et al.* [23] extend model minimization slightly to create the concepts of MDP homomorphisms and stochastic bisimilarity. These allow for the creation of more compact abstract models by allowing the mapping between the original and abstract actions to be arbitrary rather than the identity mapping.

HEXQ The HEXQ algorithm of Hengst [33] attempts to exploit the factored structure of an MDP to find repeated sub-structures and to share calculated policy information among them. In particular, it exploits how some state variables change less often than others, and how these frequently changing state variables often have the same transition dynamics for many values of the less frequently-changing state variables.

HEXQ creates a linear hierarchy of models that successively better approximate the input domain. The nth level up aggregates states based on the n most frequently changing state variables, combining the abstraction mapping determined by the next lower level with the nth most frequently changing state variable, and then creating its own abstraction. The abstraction at each level is based on the idea of entries and exits: HEXQ finds strongly connected components of state space and calculates policies, exported as abstract actions, to go from entries to exits.

HEXQ first uses sample trajectory data to order the state variables by how often they tend to change. It then creates the model for each level, starting at the bottom and working up. The model for level n in the hierarchy is initialized by taking the cross-product of model $n-1$ with the nth most frequently changing state variable (the lowest model is initialized just with the most frequently changing state variable). HEXQ calculates dynamics for the level n model using transition frequency counts from the sample trajectory data. Some actions at some states in this model cause transitions in less frequently changing state variables (which have not yet been incorporated in the model); these state-action pairs define entries and exits. Other actions at states do not cause transitions in less frequently changing state variables, and these are used to define strongly connected components of the state space. HEXQ then creates an abstraction whose states are the connected components and whose actions are policies that go to exit states and then transition to a different connected component. This abstraction is exported up to the next level of the hierarchy. At the very top of the hierarchy, the resulting abstract model is solved using Q-learning.

The intuition behind HEXQ is that it is easiest to find abstractions when working with a smaller (more aggregated) state space, and that a good strategy is to exploit isomorphically behaving regions of state space. The result is a hierarchy of models, where each model takes successively more of the domain into account and throws away as much irrelevant information as possible.

Envelope The envelope method of Dean *et al.* [14] is a dynamic state aggregation method that creates a policy over the part of the domain that currently matters, while ignoring the rest. The part currently planned for is called the envelope, and the envelope is extended (and contracted) as necessary and expedient.

The envelope method starts by finding a straight-line plan from the starting state to the goal state, using depth-first search among the most probable action outcomes. The set of states traversed in this initial plan becomes the initial envelope of states; all other states are aggregated into a distinguished *out* state, which is configured to have absorbing dynamics. The MDP consisting of the envelope and the *out* state is solved using policy iteration, and the agent starts executing. The envelope is subsequently modified in two situations: when the agent has extra time to think before needing to act, and when the agent reaches the *out* aggregate. When extending the envelope, a state is chosen from the *out* aggregate to add to the envelope; this is usually the likeliest state to be visited, which will be the current state if the agent has fallen out of the envelope. Policy iteration is performed on the new domain model, and it converges fairly quickly since the policy and value function can be bootstrapped with the corresponding values from the last round of policy iteration.

Nicholson and Kaelbling propose a variant of the envelope method [55] that also dynamically changes the currently considered state space, but does so for factored MDPs. Instead of ignoring several states at a time, it creates an abstract model by ignoring several state variables that it concludes are not relevant to the current portion of state space. All state variables are ranked using sensitivity analysis according to how they affect the state variables specifying the goal. The most affecting state variables are included in the factored envelope (i.e., in the abstract model), and the least affecting are excluded. The excluded state variables may be added later if there is extra time to think or if it is impossible to reach the goal in the current abstraction. The dynamics of each abstract state in the abstract model are assigned to be the uniform average of the aggregated states, and the abstract model is solved, as with the non-factored envelope method, using policy iteration.

State-space approximation The state-space approximation method of Boutilier and Dearden [7, 17] is similar to the factored envelope method of Nicholson and Kaelbling, in that both methods attempt to determine state variables that can be ignored by working backward from the goal state and figuring out what state variables are relevant to achieving it.

The difference is that state-space approximation requires the given model dynamics to be represented in a probabilistic STRIPS format. This more compact, structured representation allows it to deduce more quickly which state variables affect the goal most. Boutilier and Dearden also give a bound on the difference in expected reward between the optimal abstract policy and the optimal policy in the original domain.

Dynamic non-uniform abstractions The dynamic non-uniform abstraction method of Baum and Nicholson [2] is also similar to the factored envelope method. However, instead of uniformly including or discarding state variables, this method locally calculates state variable relevance and then creates the abstract domain by locally including or discarding state variables based on that information.

The dynamic non-uniform abstraction method creates the initial abstract domain aggregate states as with the factored envelope method, by aggregating only across state variables that are relevant to the goal. However, it also uses the decision tree-based representation of transition dynamics to ensure that states with different outcomes under some action are distinct and not aggregated; this causes some local disaggregation.

After the initial abstract domain is created and solved, the agent begins executing the computed policy. The dynamic non-uniform abstraction method gives some heuristics for dynamically modifying the aggregation. One test is based on the current policy and local differences in which state variables are aggregated, and this test is designed to prevent local mismatches in the aggregation coarseness from causing the agent to fail to reach the goal. Another test is based on the likelihood of reaching states, and is used both to make the aggregation finer in newly relevant parts of state space and to coarsen the aggregation in the parts left behind.

Sparse sampling The sparse sampling method of Kearns *et al.* [39] is a state aggregation method that is interesting in that its running time has no dependence on the size of the state space; instead, its running time depends on the horizon time (i.e., on the discount factor). The idea is to plan around the current state in a bubble of states large enough to guarantee being within a constant factor of optimality.

The sparse sampling method uses a generative model of the MDP. Given the task of choosing an action to take at a state, it constructs a sparse lookahead tree. This is essentially a tree of trajectories where the nodes are labeled with states, and the edges are labeled with actions and rewards; the current state is the root of the tree. The tree has a constant branching factor and is constructed by starting at a node and using the generative model to sample some number of transitions for each possible action. (The same state can appear more than once as a child of some node.) After expanding the tree to some depth, a long-term value of zero is assigned to leaves, and the values are backed up the tree to derive an estimate of taking each possible action at the current state.

Kearns *et al.* prove that, given the original model's discount factor, it is possible to choose some expansion depth and branching factor for the lookahead tree so that using the sparse sampling method will cause the agent to behave within a constnat factor of optimal. Even though this method's running time is independent of the state space size, it is still rather inefficient if the discount factor is close to one, and so various methods are proposed for making it more efficient, e.g., decreasing the branching factor further down the tree.

2.2.4 Temporally extended action approaches

A fourth category of abstraction methods consists of those that use temporally abstract actions to represent the domain. The advantage of using these abstract action methods is that they abstract temporally, and so it is possible to plan using "actions" that take a long time and traverse a large area of state space. Unfortunately, it is generally difficult to come up with useful abstract actions automatically, requiring a domain expert's intervention or a lot of training data.

Compositional Q-learning The compositional Q-learning (CQ-learning) method of Singh [75] is presented as a way of transferring knowledge between problems by packaging pieces of the solution into subtasks. The information gained from solving these subtasks can be used to bootstrap the solving process in any number of different domains.

In addition to the input domain, the CQ-learning algorithm starts out with a set of elemental tasks and a compositional task. The elemental tasks are specified as reward functions over the input domain, and the compositional tasks consists of a sequence of elemental tasks to be completed. To solve the compositional tasks, the algorithm exploits the way in which the Q functions for the compositional and elemental tasks relate. In particular, the Q function for a compositional task executing some elemental task is related to the Q function for that elemental task through addition of a term dependent only on the compositional task executing and on the position of the elemental task in the compositional task's sequence. This makes it relatively simple to solve the compositional task by first deriving the Q functions for elementary tasks and then using those to bootstrap learning the Q function of the compositional task. This especially works well when an elemental task must be executed several times in a compositional task sequence, allowing the sharing of effort within the solution of a single problem and not just across different problems.

Macro-actions The macro-action framework developed by Hauskrecht *et al.* [31] is similar to Dean and Lin's hierarchical policy construction and Parr's cached policy sets. The scenario that the macro-action framework targets is one where there are multiple related problems to be solved and so it will be advantageous to create macro-actions that help solve the domain.

As with cached policy sets, state space is broken into regions joined at boundary states, each region has some temporally extended actions calculated to go between boundary states, and an abstract model is created. (The macro-action framework creates a multi-time MDP, equivalent to a semi-MDP; see section 2.2.4.) The temporally extended actions for each region are policies created by considering the region to be a standalone MDP with boundary states being absorbing and having fixed values. Hauskrecht *et al.* describe several heuristics for building macro-actions, some that are quicker but potentially forfeit expected reward, others that need to produce more macro-actions and are therefore slower but guarantee optimality of the overall solution within some constant factor. In order to create macro-actions for the correct boundary state values, they also describe a technique that iterates between creating macro-actions and estimating boundary state values, using one to update the other.

The macro-action framework is targeted at speeding the solution of multiple related problems, not just a single problem, otherwise the time spent creating macros may not amortize over enough actual problem solving to be worth the effort. When the problem description is changed slightly, perhaps locally adjusting dynamics or switching the location of the goal, most macro-actions can be reused; generally, only the macro-actions nearby to the change in state space need to be redone.

Options The options framework of Precup *et al.* [65] uses temporally abstract actions to make value iteration be more efficient and converge in fewer time-steps.

In addition to an MDP to solve, the framework takes options as input, where options are closed-loop possibly non-Markovian policies. An option is composed of three parts: a set of states where it can act, a decision rule that says how to act, and a completion function that calculates the probability of the option terminating. Options are executed like ordinary actions from the input domain; they can be thought of as having subroutine-like semantics.

These options are added to the set of available actions in the input domain, and then the combined domain is solved using value iteration. Options may execute for more than one time-step, and this causes the normal Bellman equations to discount the action temporally in an incorrect way. In order to cause each action to be discounted according to the actual number of timesteps it is expected to execute, the combined domain is represented as a multi-time model [64]. A multi-time model represents the discounting of future rewards by scaling the transition probability function appropriately. Multi-time models are essentially equivalent to semi-MDPs.

The idea behind the options framework is to speed up the convergence of value iteration. Precup *et al.* and also McGovern *et al.* [50, 49] show that indeed Q-learning with options is considerably faster than without. This speedup comes from the way that options are faster than the original actions at propagating estimated long-term value between different parts of state space, due to the relatively large possible distance in state space between an option's invocation and termination states. (Hauskrecht notes that speedier convergence of value iteration is dependent on using an appropriate initial value function estimate [30].) Despite quicker convergence, the options framework still assures optimality since all the original actions from the input domain are still available to the agent.

Rohanimanesh and Mahadevan [71] extend the options framework to work with factored MDPs. Assuming that the MDP's state variables can be divided into subsets and that each original action or option only affects one subset of state variables, multi-options can be built by simultaneously executing several options. Rohanimanesh and Mahadevan demonstrate that, as with the original options framework, the multi-options framework gives rise to a semi-MDP, and they explore different termination conditions for multi-options.

Relativized options The relativized options framework of Ravindran and Barto [69] combines the options framework with model minimization. The idea is to exploit symmetries in the domain by using options to expedite the computation of results once, and then mapping those results to dynamically similar areas of state space.

The relativized options framework builds on Ravindran and Barto's earlier work on MDP homomorphisms (see section 2.2.3), extending it to SMDP homomorphisms. These homomorphisms exploit the way that different parts of state space may have the same dynamics. They map each of those parts of state-action space onto the same abstract model, making the correspondence explicit and giving an abstract model to work in such that any results can be unmapped back into the original parts of state space.

Given a partial MDP homomorphism ("partial" meaning that it operates on some but not all of the state space), the relativized options framework performs Q-learning in the input model. However, when in some part of state space that is mapped to the abstract model by the partial MDP homomorphism, the Q-learning (and action selection) happens in the abstract model, not in the input model. This allows the Q function information being learned in one part of state space to be used and built on in homomorphic parts of state space. Since the options are being used to speed up learning in the abstract model, Q-learning for the whole model speeds up as well.

Ravindran and Barto also briefly discuss the possibilities of using relativized options with approximate MDP homomorphisms, formalizing the results using bounded-parameter MDPs [24].

Local controllers The local controller method of Guestrin and Ormoneit [29] extends the temporally extended action methods like macro-actions and options to continuous space MDPs.

The algorithm takes as input not only some domain to solve but also local controllers. These controllers take parameters specifying exactly how they should act, e.g., what goal they should aim for. These controllers are also associated with landmarks, locations in state space near which the controllers are likely to perform well. State space is divided up into the Voronoi diagram calculated from the landmarks, with each controller being active in its landmark's subpart.

Given the domain and local controllers, the algorithm uses trial runs to estimate the transition probabilities and rewards when executing controllers in each subpart. These dynamics then allow the construction of an abstract domain with a discrete state space, where there is one state per subpart, and the allowed actions for a subpart are the various parameterizations of its local controller. Guestrin and Ormoneit demonstrate how the local controller method can be applied not only to general discounted MDPs but also to motion planning problems. In the latter case, the algorithm finds a path whose probability of failure falls below a specified threshold.

Hierarchical DG learning The hierarchical DG (HDG) learning algorithm of Kaelbling [37] is a hierarchical version of the distance-to-goal (DG) learning [38] algorithm that trades away guaranteed optimality for much better performance. DG and HDG learning operate domains where the object is to reach some goal state as soon as possible.

DG learning is similar to Q-learning, except that instead of building a Q function mapping state-action pairs to expected reward, it builds a DG function mapping state-action-goal tuples to expected reward. The DG function is learned by executing trial runs and using an analog of the Q function update equation. Since a single step from the trial run could theoretically be a single step towards any goal, the DG function can be updated for all goals at each step. This allows for efficient knowledge transfer when choosing a different goal and is the selling point of DG learning.

HDG learning does DG learning locally, but it first builds an abstract model called a landmark network. The abstract model's state space is composed of certain states from the input domain that are designated as landmarks, and abstract actions go from each landmark to its neighbors. The input domain state space is divided into subparts, one for each landmark, with each state belonging to the same subpart as its nearest landmark. HDG learns two different policy structures: first, it learns how to get from any state to any other within a subpart, and second, it learns policies to get from each landmark to adjacent landmarks. The latter data structure is all that is needed to get from one state to another if the states are in the same subpart; if the states are in different subparts, then the agent takes an action that propels it towards the adjacent landmark closest to the goal.

Due to allowing an arbitrary choice of landmarks, the HDG algorithm makes no guarantees about the optimality of the path taken from any state to the goal. However, partitioning the state space into subparts does cause the algorithm to learn faster, since common routes from landmark to landmark will be learned and reused quickly. The HDG data structures are also more compact, since the DG function only needs to be maintained for reaching local goals.

Airport hierarchy The airport hierarchy method of Moore *et al.* [52] builds on HDG, extending it in a couple of ways: the landmarks are found automatically, rather than needing to be given as inputs; and many levels of hierarchy are constructed, rather than just two. The aim of this method is still though to build *multi-policies*, i.e., a data structure that allows easy policy knowledge transfer to problems in the same domain but with different goals.

The airport hierarchy built by this method is an arrangement of the input domain's states into a network of airports. Airports have a designated seniority level, and there are exponentially fewer airports allowed at increasing levels of seniority. The network is built by successively adding airports, each time choosing the state that is farthest (in terms of expected cost) from existing airports. For a chosen state, the method designates it as an airport in the most senior level in which there is still room, and calculates policy information for reaching that state. There are two types of policy information stored in the airport hierarchy. First, for each state s, the hierarchy knows how directly to step optimally towards s from some number of states nearby. This information is calculated by using policy iteration in a gradually expanded envelope around s, bounding the expected reward results for leaving the envelope. Second, the hierarchy knows how to travel from a state to a goal if it knows how to step directly through some sequence of successively junior airports.

To act at a state, the agent uses direct policy step information if such exists for its goal. If the goal is too far away for such information to be pre-calculated, then the agent searches for the most junior airport possible such that it knows how to step towards the airport and the airport knows how to get to the goal. This process works for getting from any state to any goal and transfers knowledge efficiently between problems with different goals without being wildly suboptimal. Also, due to the exponential construction of the airport network and policy information, the space required for representing the policy is only approximately $O(n \log n)$.

Bottleneck subgoaling The bottleneck subgoaling algorithm of McGovern and Barto [48] automatically creates abstract actions by identifying bottlenecks in the state space. These bottlenecks are used to create options to speed up learning.

The bottleneck subgoaling algorithm simultaneously executes and updates its current best policy. Using the trial run data gathered so far, the algorithm identifies bottlenecks by looking for states usually visited on successful trajectories but not visited on unsuccessful ones. Using the concept of diverse density, it rates regions of state space based on how often they are visited during successful trajectories and not visited during unsuccessful trajectories. During the execution and updating, if any region shows up as consistently rated higher than its neighbors (i.e., more likely to be visited only on successful trajectories), then that region of state space is turned into a subgoal by creating a new option that is a sub-policy that attempts to reach the subgoal. The set of states over which the option can be executed are the subgoal's predecessors from the successful trial runs whose path went through the subgoal.

As with the options framework, the subgoal-achieving options created by this method speed up learning by focusing on the paths and areas of state space likely to be used in good policies.

2.2.5 Control hierarchy approaches

A fifth large category of abstraction methods has to do with controlling the domain in a hierarchical manner. There are actually several different approaches that all fall under the label of control hierarchies. For instance, some algorithms decompose the control task into subtasks, whereas others decompose the state space into subparts to be controlled separately. Also, some algorithms create a control hierarchy that sits alongside the domain, whereas others use the control hierarchy to create explicit abstract models.

Control hierarchies work well for a lot of problems because they create several levels of abstraction, so that planning can be done on a large scale, both spatially and temporally. One potential downside of control hierarchies though is that it is possible to get trapped executing in the middle of the hierarchy when the situation has changed; to combat this, several of these algorithms deliberately start at the hierarchy root and go down the hierarchy at each time-step.

Feudal RL Feudal reinforcement learning, created by Dayan and Hinton [11], uses a control hierarchy to learn how to accomplish subtasks at different temporal resolutions and in different areas of state space. This multi-resolution knowledge takes slightly longer than pure Q-learning to gain initially, but it promotes easy knowledge transfer to other tasks in the same domain.

The control hierarchy used by the feudal RL system is a tree structure successively subdividing state space, so that each node corresponds to certain states, and a node's children's states form a partition of its own states. At the top of the hierarchy, the root node corresponds to the entire state space; at the bottom, nodes correspond to just a few states. During execution, one node from each level of the hierarchy is active, and this node is the one whose corresponding set of states contains the current state.

Each node is given a set of tasks that it can request of its children. Children are given a Q-learning reward signal based on how well they accomplish the tasks that their parent gives them. This reward signal is used to learn Q functions at each node; this allows the lower level nodes to learn the correct solutions to tasks even if those tasks turn out not to be useful to the parents who asked for them. This method of distributing reward causes feudal RL to be slower in learning to solve the domain initially than regular Q-learning. However, the Q functions (and derivative policies) being learned are general and cause the feudal RL system to adjust much more quickly to a changed goal than learning to act from scratch again.

Hierarchies of abstract machines The hierarchies of abstract machines framework of Parr and Russell [57] supplements the input domain with a set of finite-state machines that constrain the agent's action choices at certain states. These machines can be combined with the input domain to form an abstract domain representing the action choices remaining to the agent.

A hierarchy of abstract machines (HAM) is composed of a set of non-deterministic finite state machines (NDFSMs); the whole hierarchy combined acts like one giant NDFSM, where only one machine is active at a time. Each machine may be in one of several states: an action-taking state, a transitioning state (where it makes a subroutine call or return to another machine), or a non-deterministic choice state. The machines are arranged in a hierarchy according to which machine makes subroutine calls to which other machines. The interpretation of the hierarchy is that its NDFSMs keep executing until they reach non-deterministic choice states, where the agent gets to choose actions. Each machine's transition function to the next machine state is a stochastic function of the current machine state and the current state of the environment. Overall, the HAM forms a set of constraints on the policy executed, but it itself is not a complete policy because of the choice states.

The HAM is combined with the input MDP model to form an abstract MDP whose states are the choice points and whose actions make choices at those choice points. The abstract MDP is solved using any normal MDP solver such as policy iteration, and any optimal solution to the abstract MDP is guaranteed to be optimal among all solutions to the input MDP that are consistent with the constraints specified by the HAM. Since the abstract MDP is non-trivial to construct explicitly (though its construction is often still advantageous), Parr and Russell also discuss a Q-learning variant that essentially learns to take actions in the reduced abstract MDP without needing to build it in full.

MAXQ The MAXQ hierarchy method of Dietterich [18] is similar to feudal RL (see section 2.2.5) except that the domain is decomposed into subtasks in such a way that no subtask ever becomes non-Markovian.

The input model is supplemented with a MAXQ hierarchy, a data structure that estimates both the context-independent and the context-dependent values of abstract actions. A MAXQ hierarchy contains alternating layers of Max nodes, which represent primitive actions or subtasks, and Q nodes, which represent actions that can be performed to achieve their parent subtasks. The difference between the two node types is that the Max nodes learn the context *independent* long-term estimated reward for performing their associated subtask, while the Q nodes learn the context *dependent* reward (i.e., how valuable is this action in achieving the parent subtask). The same Max can be used by multiple parents in the MAXQ hierarchy, allowing the value function and Q function to consolidate calculations for similar parts of the domain.

The MAXQ method uses a MAXQ hierarchy to build a hierarchical policy, which is composed of one policy for each Max node. This hierarchical policy is learned using a hierarchical Q-learning algorithm, MAXQ-Q, and the policy is executed hierarchically by Max nodes making subroutine calls to their child Max nodes.

There are two different optimality criteria that hierarchical algorithms such as MAXQ could meet. If an algorithm is *hierarchically optimal*, then it chooses the best policy that is consistent with the constraints that the hierarchical structure enforce. If an algorithm is *recursively optimal*, then each node of the hierarchy chooses the best policy given the choices of its child nodes; this gives a potentially worse solution than a hierarchically optimal algorithm with the same constraints.

Ditterich shows that MAXQ-Q converges to a recursively optimal solution and discusses the conditions under which the state representations (abstractions) chosen for the Max nodes allow for an overall representation of the optimal solution. He also shows how to execute the policy non-hierarchically for increased performance where the recursively optimal solution is not the overall optimal one.

ALisp The ALisp framework of Andre and Russell [1] defines a Lisp-like language for specifying constraints on agent action choices. ALisp is similar to the HAM framework (see section 2.2.5) in that it enumerates actions to take at certain states, choices for the agent to take at others, and subroutine calls. The ALisp constraints are used to create a hierarchical structure representing an abstract semi-MDP whose actions and states are the choices an agent can make and the locations at which those choices occur. The abstract model is solved using an ALisp-specific variant of MAXQ learning called ALispQ.

Andre and Russell show how ALisp allows a three-part decomposition of the domain's value function that improves upon MAXQ's two-part decomposition, in that it permits a certain type of abstract state representation without sacrificing hierarchical optimality. This improved allowance for abstract state representation at various points in the algorithm greatly improves its performance over regular Q-learning due to the sharing of learned Q function information between different parts of the domain that call the same subroutine.

Task hierarchy The task hierarchy method developed by Pineau *et al.* [60, 61] uses a set of hierarchical control constraints to derive state abstractions. Each task performs policy-contingent abstraction, examining its subtasks' policies to figure out how best to abstract the domain.

In addition to an input domain, the task hierarchy method requires a task-based hierarchical decomposition. Each task, starting with the task at the top of the hierarchy to solve the domain, is decomposed into the subtasks (which can be shared) and primitive actions that might be necessary to complete the task. Each subtask is considered to be an abstract action available to its supertask(s). In addition to having a set of available abstract and primitive actions, each task has a set of goal states along with a function giving their relative desirability.

Given a domain and task decomposition, the algorithm creates an abstract version of the domain for each task, from the bottom up. This abstract version is calculated using model minimization, where the model to be minimized is the input domain where the original actions are replaced by this task's allowed actions and where the original reward function is augmented with the task's goal-oriented reward function. The calculated abstract domain for a task is solved, and the resulting policy's estimated dynamics are noted by any supertasks. After an abstract domain and policy are created for each task, the hierarchy is executed by tracing from the root to a leaf at each time-step, choosing at each level whichever action promises to be best.

The task hierarchy method improves upon normal model minimization in that it performs model minimization for different tasks, and presumably different areas of state space, separately. This allows it to exploit how certain aspects of the domain may be irrelevant for some task though they are very relevant later. This method also weeds out irrelevant domain features by creating the abstract domains from the bottom up: the abstract domains at higher levels need only represent features relevant to the actual policies that will be executed at lower levels, rather than any features that could possibly be relevant.

2.3 Comparison of previous MDP methods

The abstraction methods are grouped above into one possible categorization, but there are many other possible categories and many similarities between individual methods. For instance, cached policy sets uses abstract actions despite being in the decomposition and recombination category, and several algorithms like the airport hierarchy and bottleneck subgoaling automatically create subgoals.

To get a better idea of what each method is like than just its main strategy, tables 2.1, 2.2, and 2.3 summarize different characteristics of each.

Table 2.1 gives information about the sort of inputs each algorithm needs. Some methods just need a simple MDP; other methods need or can take advantage of the model being structured in some way; others need additional domain-specific information.

- The column *factored input* indicates whether the method can take advantage of an input model expressed as a factored MDP (as opposed to an MDP without state and action spaces as variables), and/or whether such a factored input model is actually required.

- The column *discrete/continuous input* indicates whether the method operates on discrete or continuous state and action spaces. A method may have variants that allow it to operate on both discrete MDPs and on continuous MDPs; it may also operate on hybrid MDPs, which contain both continuous and discrete state and action variables.

- The column *full or generative model* indicates whether the method needs to be

Method name	factored input	discrete/continuous input	full or generative model	structured input dynamics	multiple input models	type of additional input
Airport hierarchy	not req.	discrete	generative	unstructured	single	none
ALisp	not req.	discrete	generative	properties	single	control constraints
Bottleneck subgoaling	not req.	discrete	generative	unstructured	single	none
Cached policy sets	not req.	discrete	full	unstructured	single	state space partition
Compositional Q-learning	not req.	discrete	generative	unstructured	multiple	subgoals
Distributed hierarchical planning	required	discrete	full	unstructured	single	state variable partition
Dynamic MDP merging	not req.	discrete	full	unstructured	multiple	none
Dynamic non-uniform abstractions	required	discrete	full	decision tree	single	none
Explanation-based RL	required	discrete	full	PSTRIPS	single	none
Envelope	used	discrete	full	unstructured	single	none
Factored value functions	required	discrete	full	additive reward fn.	single	basis functions
Feudal RL	not req.	discrete	generative	unstructured	single	control constraints
Hierarchical decomposition of goals	not req.	discrete	generative	unstructured	single	subgoals
HEXQ	required	discrete	generative	unstructured	single	none
Hierarchical policy construction	required	discrete	full	unstructured	single	state space partition
Hierarchical SMART	not req.	discrete	full	unstructured	multiple	none
Hierarchies of abstract machines	not req.	discrete	generative	unstructured	single	control constraints
Local controllers	not req.	continuous	full	unstructured	single	abstract actions
Logical Q-learning	required	discrete	full	PSTRIPS	single	none
Macro actions	not req.	discrete	full	unstructured	single	abstract actions
Markov task decomposition	required	discrete	full	unstructured	multiple	heuristic
MAXQ	not req.	discrete	generative	unstructured	single	control constraints
Model minimization	used	discrete	full	unstructured	single	none
Multigrid value iteration	required	continuous	full	unstructured	single	none
Options	not req.	discrete	full	unstructured	single	abstract actions
Prioritized goal decomposition	required	discrete	full	additive reward fn.	single	none
Relativized options	required	discrete	full	unstructured	single	mapping,abstract actions
Sparse sampling	not req.	discrete	generative	unstructured	single	none
SPUDD, APRICODD	required	discrete	full	ADD	single	none
State-space approximation	required	discrete	full	PSTRIPS	single	none
Structured policy iteration	required	discrete	full	decision tree	single	none
Task hierarchy (PolCA)	required	discrete	full	unstructured	single	control constraints
UTree and TTree	required	both	full	unstructured	single	none

Table 2.1: Input characteristics of previous methods.

fed the full MDP with dynamics and reward functions as input, or whether it simply uses some sort of generative model to sample the dynamics and reward functions.

- The column *structured input dynamics* indicates whether the method requires the input dynamics to be structured in some way. For example, the structuring may involve representing functions using algebraic decision diagrams or probabilistic STRIPS-like rules. Generally, if a method requires structured input dynamics, then it also must require an input domain with factored state and action spaces.

- The column *multiple input models* indicates whether the method actually operates on multiple separate MDPs with separate dynamics. (The MDPs are linked somehow else, perhaps through the agent trying to maximize the sum of all rewards.)

- The column *type of additional input* indicates whether the method takes any additional input beyond the (possibly structured) input model, and, if so, what form that input takes. This category only contains model-dependent inputs and specifically excludes parameters that would be the same for every model, e.g., a learning rate.

Table 2.2 gives information about the sort of output each algorithm gives. The methods give output ranging from a new, hopefully simpler, abstract domain to a complete policy, and everything in between. Table 2.2 also gives information about the running characteristics of methods (like being anytime, or planning offline vs. online) and the optimality they can or must guarantee.

- The column *end goal* indicates whether the method is attempting to solve the input domain (e.g., by creating a policy for it), or whether it creates an abstracted version of the domain. Some of the methods are listed as abstraction

Method name	end goal	goal or continuous	end product	dynamism	approximation control	level of optimality
Airport hierarchy	solve	goal	Q fn.	convergent		none
ALisp	abstract	continuous	SMDP	convergent		hierarchical
Bottleneck subgoaling	abstract	continuous	MDP	convergent		exact
Cached policy sets	abstract	continuous	SMDP	static	●	within epsilon
Compositional Q-learning	solve	goal	Q fn.	convergent		recursive
Distributed hierarchical planning	solve	continuous	policy	static		other
Dynamic MDP merging	solve	continuous	value fn.	static		exact
Dynamic non-uniform abstractions	solve	goal	policy	dynamic	●	none
Explanation-based RL	solve	goal	value fn.	convergent		exact
Envelope	solve	goal	policy	dynamic	●	within epsilon
Factored value functions	solve	continuous	policy	static		within epsilon
Feudal RL	solve	continuous	Q fn.	static		recursive
Hierarchical decomposition of goals	solve	goal	Q fn.	convergent		recursive
HEXQ	solve	continuous	Q fn.	convergent		recursive
Hierarchical policy construction	abstract	continuous	MDP	static	●	exact
Hierarchical SMART	solve	continuous	Q fn.	convergent		none
Hierarchies of abstract machines	abstract	continuous	MDP	static		hierarchical
Local controllers	abstract	continuous	SMDP	static		none
Logical Q-learning	solve	continuous	Q fn.	convergent		exact
Macro actions	solve	goal	SMDP	static	●	recursive
Markov task decomposition	solve	goal	value fn.	dynamic		none
MAXQ	solve	continuous	Q fn.	convergent		recursive
Model minimization	abstract	continuous	MDP	static	●	exact
Multigrid value iteration	solve	continuous	value fn.	convergent		within epsilon
Options	solve	continuous	value fn.	static		exact
Prioritized goal decomposition	solve	goal	policy	static		none
Relativized options	solve	continuous	Q fn.	static		exact
Sparse sampling	solve	continuous	policy	dynamic	●	within epsilon
SPUDD, APRICODD	solve	continuous	value fn.	static	●	exact
State-space approximation	solve	goal	policy	convergent	●	within epsilon
Structured policy iteration	solve	continuous	policy,value fn.	static		exact
Task hierarchy (PolCA)	solve	continuous	value fn.	static		recursive
UTree and TTree	solve	continuous	policy	static	●	exact

Table 2.2: Output characteristics of previous methods.

methods, even though in published papers they always appeared in combination with some MDP solution method like value/policy iteration or Q-learning. If the method creates one new MDP, though, and if that MDP is clearly an abstract version that could be abstracted further, then it is categorized as creating an abstract version of the domain. This classification is useful for identifying existing methods that can easily be combined with each other.

- The column *end product* indicates what sort of result the method produces. In the case of methods whose end goal is to solve the input domain, then this can be a complete policy, a Q function, or a value function. A method may come up with more than one of these, and they are obviously reconstructible one from another; this column simply describes the immediate end product of a method. In the case of methods whose end goal is to create an abstracted version of the domain, this column describes the format of the output model.

- The column *dynamism* indicates how the abstraction method behaves while the agent is executing in the domain. A method may be static and completely formulate its policy/abstraction before execution begins. A method may use information gained during execution to refine its policy/abstraction, converging on some solution. (Static methods may also converge to a solution, but methods labeled *convergent* have no specified stopping criteria given by the authors or may never actually reach a final iteration.) Finally, a method may continually change its policy/abstraction during the course of execution, never approaching any fixed solution.

- The column *approximation control* indicates whether there is a separate parameter that can adjust the level of approximation that this method uses. This must be a separate parameter, i.e., it is not enough that changing the parameters that a method is given changes how approximate the result is, nor is it enough that a convergent method can be stopped early.

- The column *level of optimality* indicates what type of optimality a method can guarantee, if any. The types of optimality are exact, hierarchical, and recursive. In exact optimality, the method guarantees that the expected long-term discounted reward is the same as solving the input domain optimally. Hierarchical and recursive optimality are defined by Dietterich [18] and have to do with methods that decompose a domain hierarchically: hierarchical optimality guarantees optimality among all policies consistent with the hierarchical domain decomposition, while recursive optimality guarantees optimality within each subtask viewed individually.

Table 2.3 gives information about the primary approach that each algorithm takes and how it operates on the domain.

- The column *abstraction type* indicates which of the five broad abstraction types described above the method appears in.

- The column *state abstraction* indicates whether the method performs some sort of abstraction on state space during planning or execution.

- The column *temporal abstraction* indicates whether the method performs some sort of temporal abstraction during planning or execution. This involves explicitly creating a model or actions or something that is temporally extended.

- The column *dynamics abstraction* indicates whether the method creates abstract dynamics during planning or execution. The dynamics are an abstract version of some input dynamics, not just simple things like absorbing states.

- The column *abstract actions* indicates whether the method uses abstract actions during planning or execution.

- The column *subgoals* indicates whether the method creates subgoals during planning or execution. The subgoals are explicit in that there are actions or sub-policies created for the agent to execute in order to reach the subgoal.

Method name	abstraction type	performs state abstraction	performs temporal abstraction	performs dynamics abstraction	creates/uses abstract actions	creates/uses subgoals	creates/uses a control hierarchy	joins pieces	ignores features	uses symbolic reasoning
Airport hierarchy	temporally extended actions	●			●	●				
ALisp	control hierarchy	●					●			
Bottleneck subgoaling	temporally extended actions		●	●	●	●				
Cached policy sets	divide and conquer		●	●	●		●	●		
Compositional Q-learning	temporally extended actions		●		●					
Distributed hierarchical planning	divide and conquer							●		
Dynamic MDP merging	divide and conquer							●		
Dynamic non-uniform abstractions	state aggregation	●		●					●	
Explanation-based RL	structure	●								●
Envelope	state aggregation	●		●					●	
Factored value functions	structure	●								
Feudal RL	control hierarchy		●	●	●		●			
Hierarchical decomposition of goals	temporally extended actions	●	●	●	●	●				
HEXQ	state aggregation	●		●	●		●		●	
Hierarchical policy construction	divide and conquer	●	●	●	●		●	●		
Hierarchical SMART	divide and conquer		●		●		●	●		
Hierarchies of abstract machines	control hierarchy	●					●			
Local controllers	temporally extended actions	●		●				●		
Logical Q-learning	structure	●								●
Macro actions	temporally extended actions	●	●	●	●		●			
Markov task decomposition	divide and conquer							●		
MAXQ	control hierarchy	●		●			●			
Model minimization	state aggregation	●		●						
Multigrid value iteration	state aggregation	●								
Options	temporally extended actions		●	●	●					
Prioritized goal decomposition	divide and conquer							●	●	●
Relativized options	temporally extended actions	●	●	●	●					
Sparse sampling	state aggregation	●								
SPUDD, APRICODD	structure	●								
State-space approximation	state aggregation	●		●					●	●
Structured policy iteration	structure	●								
Task hierarchy (PolCA)	control hierarchy	●		●	●	●	●	●		
UTree and TTree	state aggregation	●								

Table 2.3: Information about the strategies employed by previous methods.

- The column *control hierarchy* indicates whether the method creates or uses a control hierarchy of at least two levels in the course of solving or abstracting the input MDP. A control hierarchy is considered any structure where the responsibility for choosing agent actions passes up and down some hierarchy, whether that hierarchy is linear, tree-based, or something else.
- The column *joins pieces* indicates whether the method involves taking multiple domains or multiple pieces of a single domain and joining them together.
- The column *ignores features* indicates whether the method ignores features of the domain's state and/or action spaces in order to expedite planning or execution.
- The column *symbolic reasoning* indicates whether the method performs any symbolic reasoning.

Figure 2-1 lays out the prior methods, grouped by their major type, but it also connects methods that are related in some other way–perhaps they end up using similar strategies even though they approach a domain in different ways, or one uses the other as a subroutine, or one improves on the other.

The module hierarchy system created in this thesis is compared to these previous methods in detail in section 7.1.

2.3.1 Discussion of previous methods

Despite the large quantity of previous methods that attempt to solve large MDPs, all are based on one or more of just a few simple ideas about how to make the domain simpler to solve.

Divide-and-conquer In the divide-and-conquer idea, the domain is divided into smaller pieces, each piece is solved separately (whatever "solved" means), and then the

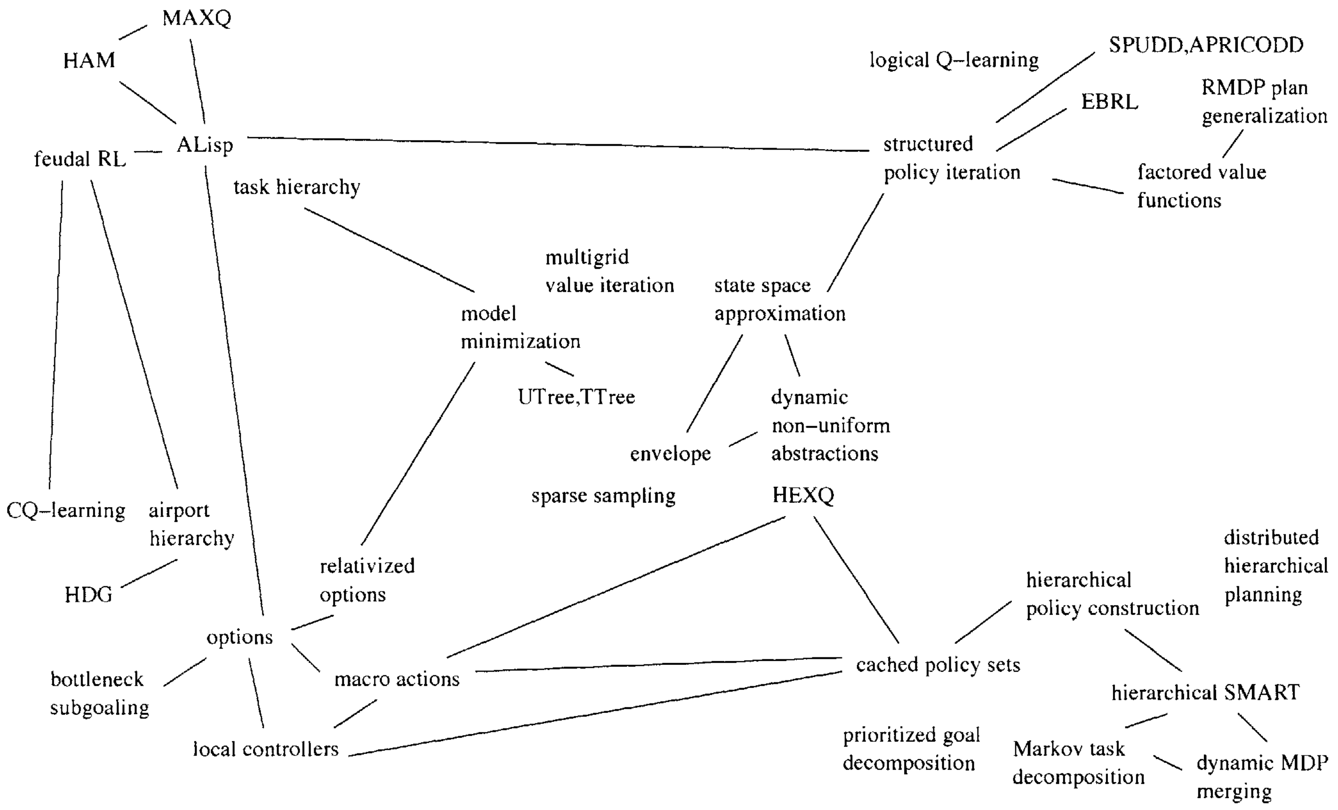


Figure 2-1: Prior methods grouped by major strategy and with closely related methods connected by lines.

pieces are reassembled, perhaps requiring some more solving. The variation between previous methods involves exactly how the division and then reassembling is done. Some methods straightforwardly subdivide the state and/or action spaces into clusters of states and/or actions. Other methods subdivide the state and/or action spaces by partitioning the state and action variables rather than the states and actions themselves. Sometimes the pieces overlap, and sometimes they don't.

The divide-and-conquer approach is successful at allowing methods to solve larger domains than would otherwise be possible because the difficulty of solving a domain increases more than linearly with its size. (In particular, creating an optimal policy for an MDP takes work polynomial in the size of the domain, as can be seen from the linear programming representation of MDPs. The policy iteration method used in practice is exponential in worst case but generally has better execution characteristics.) Since it is more difficult to solve a domain of size kN than k domains of size N, it makes sense to try to break the domain into as many small pieces as feasible.

The tricky part of divide-and-conquer is not so much dividing the domain up as it is putting it back together again. The several pieces that the method creates usually relate to each other using some sort of hierarchy. In some methods, the hierarchy is only two levels (inside each piece, and how the pieces are connected), and these approaches create something that feels like a topological map or set of connections between the pieces. Other methods create a many-leveled hierarchy, where each level appears like a successively more abstract version of the domain. In either case, the higher levels of the hierarchy tend to be some MDP-like models in their own right, and these MDPs need solving after the individual pieces are solved. Generally, having a lot of pieces tends to make the model one step up the hierarchy be large; if there are only two levels to the hierarchy, then this necessitates tradeoffs and possible compromises about the size of pieces.

Aggregation In the aggregation idea, the domain is divided as above into pieces; however, instead of giving each piece MDP-like structure in its own right, the pieces are considered more as indivisible units, where all members are treated the same. Aggregation is generally performed on the state space, either choosing individual states to cluster together, or by exploiting the state-variable structure of the state space to perform wholesale clustering across the entire space. The cluster is given dynamics that are some sort of combination of the dynamics of the individual components of the cluster.

The aggregation idea is successful at allowing larger domains to be solved, obviously, because it reduces the size of those domains. However, what is interesting is the reasoning behind why aggregation should produce an abstract domain that is a faithful representation of the original domain. In particular, the reasoning is that the differences between certain states are irrelevant as far as constructing a policy and acting in the domain go. The differences may just be currently irrelevant (i.e., the states are in a wholly different part of state space than the current state and are unlikely to be reached in the near future), or the differences may be intrinsic and due to similar transition and reward dynamics.

The tricky part of aggregation is knowing how to cluster, i.e., knowing which differences matter and which don't. This is certainly not as easy as it sounds: aggregating just two states together, for instance, may cause catastrophic loss of expected reward, even if the aggregated states are distant from any non-zero reward.

Temporally extended actions The idea of temporally extended actions is that there are often sub-policies that might be worth following one or more times over the course of solving a domain. These sub-policies might be as simple as a linear sequence of actions specified from a particular starting state, or they could be as complicated as a full-blown Markovian policy that can be active over a large area of state space. Though the sub-policies theoretically could take any form, they are generally specified in some compact form, e.g., the attainment of a particular goal state or the execution of a simple finite state machine.

These temporally extended actions have the benefit of reducing the effective size of the decision space that an agent has to search: the agent only has to make choices when temporally extended actions end. (Or, if it can interrupt temporally extended actions, then it at least has a presumably not-too-bad default action at each time-step.) A domain can be solved well using temporally extended actions if its best policy can be covered (or almost covered) by several compactly specified sub-policies. Such abstraction methods exploit that this is true of most real-world domains that are interesting to solve.

The tricky part of temporally extended action methods is what sub-policies to choose. Though the space of compactly-specified sub-policies is much smaller than the whole policy space, it is still much too large to search exhaustively, and so abstraction methods currently use simple heuristics or human-provided input in the selection of temporally extended actions.

Constraints The idea of using constraints to help abstract and solve an MDP is exactly what it sounds like: policies are only considered subject to certain constraints on the allowable actions. Of course, any abstraction method can be viewed as simply imposing constraints on the allowable policies, some more easily than others. These methods take an MDP and use a set of constraints to search through just a small part of policy space.

Like temporally extended actions, the use of constraints reduces the decision space that the agent needs to search. In particular, it is supposed to reduce the search space to a space that (a) contains an optimal or near-optimal policy, and (b) is relatively efficient to search through. Requirement (b) is important, because it makes no sense to specify a search space without having a corresponding algorithm that can search it easily.

The tricky part of using constraints is deriving good constraints. Since constraints cut out portions of policy space from consideration, those removed portions had better not contain all optimal policies. The constraint language also needs to be general enough to apply to any domain yet specific enough to have a good algorithm for searching constrained policy space.

2.3.2 Optimality vs. tractability

These previous methods have two ways that they can approach the issue of optimality. One way insists that, no matter what parameters are given, the solution in the abstract space should be within some factor of optimal. For such a method to be able to solve a very large domain, that domain must contain the particular type of structure that the method wishes to exploit. If it does not, then the method will operate very slowly, most likely sitting at the initial policy computation step for an unreasonably long time.

The second way of approaching optimality focuses more on tractability. Such methods may provide optimality within the supplied search parameters, but their focus is more on avoiding being stuck on the initial policy computation step: they willingly return a poor solution faced with the alternative of returning no solution at all. Methods based on heuristics and human-supplied parameters tend to fall in this latter category.

Chapter 3

Motivation

Each previous attempt at solving large domains has some good features. Some guarantee optimality, while others garner large reductions in model size by sacrificing optimality. Some require no extra input beyond the input domain, while others use extra input to take advantage of domain structure that might otherwise remain hidden. Some might require trial runs through the domain, while others do all calculations up front.

For each previous approximation and abstraction method, there are certain domain characteristics that the method works best with, and each method explicitly or implicitly makes certain assumptions about the domain structure available to be exploited. For instance, abstract action methods work best when the domain has relatively few meaningful choice points and when there are relatively easy ways of getting from one choice point to the next. These abstract action methods also assume either that the domain is small enough to have a domain expert create the abstract actions by hand, or that the domain is simple enough for the abstract actions to be found automatically.

The problem with very large domains is that, unless they happen to be the sort of uniform and inherently difficult domains like the stochastic traveling salesman problem, these very large domains rarely exhibit the same structural characteristics everywhere. Therefore, approximation and abstraction methods are unlikely to find their desired characteristics present in more than a fraction of the domain.

3.1 Desirable features for domain solvers

The abstraction methods presented above are a big step beyond the simple brute force approaches of value and policy iteration. They are not perfect, however, and there are some common threads that combine to make them unsuitable for use with very large domains.

Note that the deficiencies that are discussed here don't have to do with the previous methods achieving sub-optimal performance on certain domains. Generally, these methods will analyze a domain and act in a way superior to what a human could do; the only problem is that they never stop analyzing the domain and start acting, because the pre-processing step takes far too long on very large domains.

3.1.1 Complete policies vs. plans

One reason previous methods fail to scale up is that they generally create complete policies; i.e., they create state-to-action mappings whose domain is the whole of the state or belief space. In most MDPs, however, the states actually visited are a tiny fraction of all possible states. It seems wasteful to create policies that cover vast areas of state or belief space that will never be visited.

The usual reason that a complete policy gets created seems to be that it is deemed necessary to think about what happens when any possible sequence of events occurs, no matter how unlikely and no matter how unrewarding the possible path may be. It really is impossible to avoid thinking about every part of the state space at least once (assuming a long enough horizon), to make sure that all possible reward or cost gets taken into account. But it should be possible to take shortcuts, make approximations, and use information built into the model to avoid having a plan on hand for every single state or belief.

One way that some previous methods avoid part of the tedium of creating a complete policy is by creating it implicitly. That is, they create some sort of structure that encodes the complete policy in compressed form and can be queried as needed. For instance, the abstract action methods that create one or more levels of abstract semi-MDPs all store their complete policy abstractly. To find the action to take at some state, they first find the abstract action to take at that state, then ask the abstract action what concrete action it recommends.

(Note, however, that this isn't quite a complete policy, since only certain states can be represented in the abstract semi-MDP, and it is only at those states that you can start following the policy. On the other hand, the task hierarchy method represents all states in its top level abstract model, and it can therefore give an action for any state; from the way that the top level abstract model is built, though, it seems that it will have the same state space as the original domain for most non-trivial domains.)

Why not just create an implicit policy for the entire state or belief space and be done with it? The problem is that even the implicit complete policy may be too difficult and time-consuming to calculate, because all of state space needs to be checked. Consider, for instance, an intelligent agent guiding a vehicle through enemy terrain. It may be expecting to encounter up to ten of several different types of enemy vehicle or installations, in any combination. Clearly all these combinations are too much to plan for, and only a few will actually be encountered.

The envelope methods take a different approach to avoiding complete policies. They don't create implicit complete policies; instead, they assert that only the most probable parts of state space near the current estimated path need to be considered and planned for. When the current state wanders outside the envelope or when the current policy can be seen to be unable to attain the goal, the envelope is extended to encompass new information. This sort of approach, creating a plan and repairing it as needed, seems to be the only one that will work on very large domains.

3.1.2 Planning at different granularities

The existing envelope methods have some room for improvement. For instance, the envelope methods consider and create a plan for each atomic state from the starting state all the way through the goal state, rather than thinking about the future approximately and leaving its planning for the future. But it is worth thinking some more about the idea that the envelope algorithm has when it tries to use computation time on the domain parts that matter.

It seems like it should be possible to cut out policy parts that are no longer needed, and to avoid fine-grained pre-planning from start to finish. This comes from the general observation that most of the domain is irrelevant at any one time, and the particular part that is relevant is situation-dependent. For instance, when going grocery shopping, the shopping list is irrelevant while driving to and from the store, but driving skills are irrelevant while in the store.

Prior methods don't seem to take advantage of this locality of relevancy. They require that a plan be formulated for purchasing items in the store before even beginning the trip to the store. As an example, some previous methods (e.g., abstract actions) may dynamically ignore parts of the abstraction that they have created, focusing attention on what matters. But this involves pre-calculating the dynamics and then ignoring them at run-time until they are needed. In the shopping example, both a driving and a purchasing policy would be created before setting out, and then whichever is currently irrelevant would be ignored.

Not only would the purchasing plan be created needlessly early, but it might actually turn out to be wasted effort if the situation at the store is different than expected. For instance, the store could be closed because of a holiday, or the taco sauce could be in the ethnic foods section instead of with the condiments. (It is possible to make a policy that can handle all these variations, but, as already discussed, plans are preferable to policies wherever possible. It is also of course possible to make a plan that handles all the likeliest expected situations. If there is a lot of uncertainty in the domain, however, then a significant fraction of the domain will fall into the category of likely situations; if the domain is large, creating a plan that covers a significant fraction of the domain will be just as intractable as creating a policy.)

In order to take advantage of different parts of the domain mattering in different situations, it makes sense to model different parts of the domain with more or less granularity based on whether they are currently more or less relevant. The dynamics and plan for less relevant parts should be estimated only in as much detail as is necessary for the near future to be properly handled. The irrelevant parts should be temporally and spatially abstracted so that they can be properly handled with as little consideration as possible. This allows the system to begin acting sooner, and it avoids the wasted effort that would go into planning for the uncertain far future.

3.1.3 Dynamic representation changes

If the system will sometimes represent a part of the model approximately and sometimes in much detail, then it must be able to change the representation on the fly. As mentioned above, there are several different ways that this could be done. It could be done simply by having the abstraction structure remain the same but by paying attention to different parts of the domain at different times (giving computation time to certain parts as opposed to others). Alternately, dynamically changing the abstraction could mean actually refactoring the abstraction structure, using a different representation of the domain than before.

The attention-focusing interpretation of dynamically changing the representation is what some previous methods can be thought of as doing. For instance, using local controllers as abstract actions allows for the creation of an abstract semi-MDP. Then, when an abstract action is chosen, the corresponding controller might only consider the part of the state relevant to achieving that particular subgoal. Of course, this is dependent on the human domain modeler crafting the local controller in a certain way, but it is certainly possible.

Recall, though, that just resorting to attention-focusing is problematic because it insists on pre-planning everything. If the system is not going to pre-calculate all possible representations, then it must calculate some appropriate representations dynamically, while the system is running. Unfortunately, none of the previous methods does dynamic refactoring of its abstraction. The processing model they use is that information about how to abstract the domain is either given as input (e.g., abstract actions) or calculated in a pre-processing phase (e.g., model minimization); in either case, the abstraction structure is calculated and then left unchanged while being used. The envelope methods are the one exception, doing dynamic refactoring in that they only add additional states to and remove irrelevant states from the current envelope.

Of course, in a sense, dynamic re-abstraction can be done with any method: the method just needs to be run again from the beginning but with the new data. This isn't so desirable, though, because it would discard all of the previous abstracting work. This is sometimes necessary, but sometimes the data may have changed just slightly in such a way that use could be made of most of the previous abstraction. However, no previous method is able to do this.

3.1.4 Other reasons to change representation dynamically

Besides supporting changing the granularity to allow focusing attention on relevant domain parts, there are several other reasons that it would be nice to be able to change the abstraction dynamically.

One such reason is that the current plan should be reconsiderable at any time, even when the current state isn't abstractly representable. With the abstract action methods that create an abstract semi-MDP, the system can't re-plan in the middle of executing an abstract action, because there is no way for the abstract model to represent the current state. It seems like changing the representation should be possible, and it should even be relatively fast, since most or all of the dynamics that the abstract model represents would still be valid after the re-planning.

Another reason to have dynamic representation changes is that it would let you change properties of the domain for any reason you wanted. For instance, you might decide that your goal is now different, or you have better estimates of the domain dynamics. It would be better just to slightly modify the existing representation rather than to discard the calculations already done and start from scratch.

A third reason to dynamically change the representation is that the available computational resources have changed. If some equipment fails, or if other equipment is brought online, then it would be advantageous to be able to change the representation, perhaps in order to maintain real-time guarantees, or to plan at a finer resolution than was possible before.

3.1.5 Learning

A final reason to be able to change the abstraction dynamically is that it would allow for learning. If a model of the domain is not available, then the system must learn one while it is acting. This could involve only parameter learning, where things like the transition and observation probabilities are updated according to the currently guessed domain structure, or it could involve structure learning, where state variables and transition dependencies are added and removed. In either case, any abstraction built on top of the ever-changing domain should be able to salvage parts of its representation when unrelated changes occur.

Not only might the domain be learned at run-time, but the abstraction may also be learned at run-time. For instance, a new abstract action or a new subgoal may be deemed useful. This new way to abstract could be noticed either due to a change in the domain model (as discussed above) or simply due to having collected more sample-run data through acting in the domain. In either case, a new abstract action or subgoal should be easily added to the existing bunch without requiring recalculation from scratch.

Being able to learn the parameters of previous methods would be very advantageous. Some previous methods don't actually require extra input aside from the domain model (e.g., model minimization), but the more powerful ones do. Being able to find good parameters automatically would allow those abstractions to be used without needing a human to sit down, analyze the domain, and come up with a list of parameters that he thinks will work well. The process of coming up with these parameters (for action hierarchies or abstract actions, for instance) is a somewhat trial-and-error process: the system is run with a set of parameters, the output is examined, the parameters are adjusted as needed, and the cycle repeats. Trial and error is the sort of thing that computers are good at, since they never get bored and have a great memory for what worked and what didn't, so it makes sense to let the computer do at least part if not all of this work.

3.1.6 Combining multiple approaches

When trying to figure out which pre-existing abstraction method to use for a given domain, often different approaches work for different parts. For instance, it could be that one part is abstracted very well using options, but there aren't really any good options for the rest of the domain; the rest might divide up nicely, though, to be solved easily by distributed hierarchical planning. It would be nice to be able to mix and match abstraction methods as desired.

This can be currently done by patching together different methods by hand, but the interaction between the different abstractions might cause some difficulties. For instance, when creating a plan rather than a policy or when using an envelope method, the domain model exported from one abstraction module to another might change, and so the other modules must know how to deal with dynamic representation changes (as addressed in above). In other cases, it is not clear how to use an abstraction method to process just part of a domain, since it is used to operating on all of it.

Even if all the implementational difficulties were overcome, it would also be nice, when combining abstraction methods, to know what can be said about the combined structure's characteristics like optimality and running time. Unfortunately, there is currently no theory about combining different abstraction methods.

Being able to combine multiple abstraction methods can allow the weaknesses of certain methods to be offset by the strengths of others. For instance, the mixture of options and distributed hierarchical planning allows the options to perform temporal abstraction while the distributed hierarchical planner spatially abstracts the domain into segments that are mostly solvable separately.

Being able to combine multiple abstraction methods also allows for the creation of certain "abstraction methods" that wouldn't normally stand on their own but would be considered add-ons to other "real" methods. For instance, a certain state variable might be irrelevant for most of the time that the system operates, or perhaps a certain state is unlikely to occur. If abstraction methods could be combined, then it would be possible to have an abstraction method that removes the offending item until it is actually relevant. (This would rely on the subsequent abstraction methods supporting dynamic refactoring, of course.)

To make all abstraction methods work together, they would have to support things like dynamic re-abstraction and operating only on part of a domain. By doing this, the "silver bullet" abstraction methods could be replaced with a sort of component architecture.

3.1.7 Summary

There are several ways that previous abstraction methods could be made better in order to handle large domains. For instance, rather than creating an explicit or implicit complete policy, the system could create a plan and repair it as needed, like the envelope methods do. Also, since only a subset of the domain is relevant at any one time, it makes sense to abstract different parts of the domain to different granularities based on relevance.

The strategies of creating plans and abstracting to different granularities are only possible if the abstraction can be changed on-line. This ability to change the representation dynamically, when implemented as actual online refactoring rather than switching between pre-manufactured abstractions, enables some other desirable behaviors. For example, it allows the domain's parameters to be changed while the system is running.

It also enables learning, in two ways: it allows the system to adapt as the domain's structure and parameters are learned, and it allows the system to find good parameters to the abstraction that it uses. In fact, it even allows the system to change the type of abstraction it is doing, but in order to know when to change the abstraction, there must be some way of measuring how effective an abstraction is.

Finally, since different parts of domains are often structured differently, it would be nice to use multiple abstractions together on the same problem. Turning them into components, and perhaps adding learning and dynamic re-abstraction capabilities, would help create an easy-to-use, powerful system for solving large MDPs.

Chapter 4

Module hierarchy framework design

Various abstraction and approximation methods for MDPs, explored by prior researchers, have made advances and work well in certain situations, but not one is able to deal with all types of structure or all types of domain. In order to have a system that can work well in all situations, something different is needed that combines different abstraction and approximation methods and dynamically changes the abstract domain representation to focus on the currently relevant portions. This section describes one possible such system, a *module hierarchy framework*.

4.1 Gridworld example

To understand the module hierarchy framework, it will be useful to have a running example. Consider a trivial example of a robot that lives in a 10×10 gridworld (see figure 4-1(a)). The robot can carry packages, and its goal is to pick them up at one location and drop them off at another. The robot's movement is stochastic, so that with some small probability it fails to execute the action it attempts, or it moves in the wrong direction. The robot has a battery whose charge gradually runs down

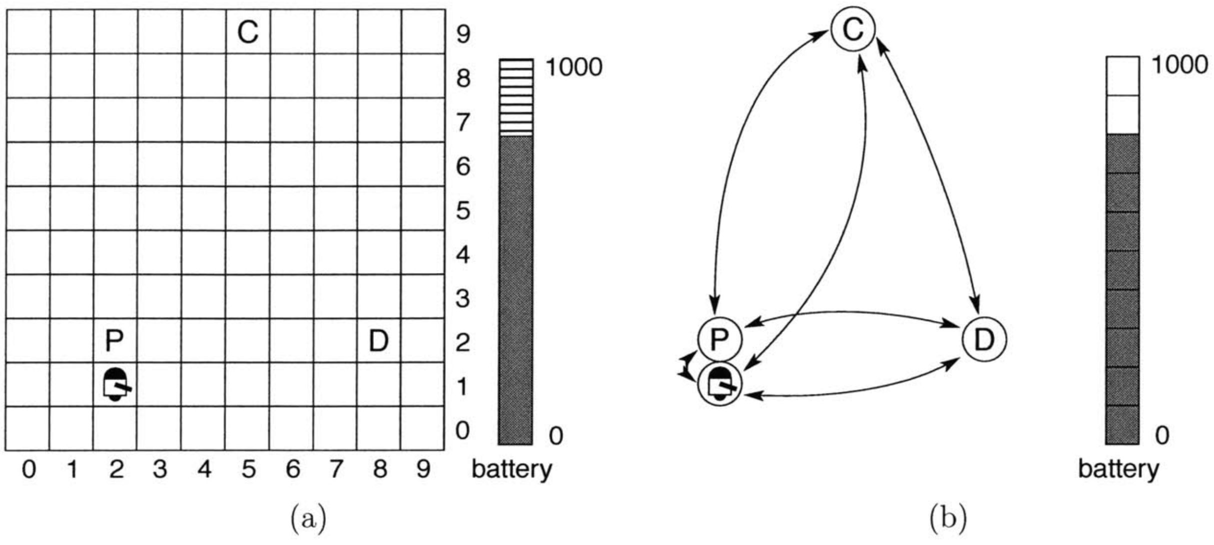


Figure 4-1: (a) The package and charger gridworld example. (b) An abstract version of the gridworld example.

from 1000 to 0 and needs to be charged to full periodically at a charger. Finally, the robot gets a reward, discounted over time, for successfully transferring packages. This domain has $10 \times 10 \times 2 \times 1000 = 200,000$ states and seven action values (*north*, *south*, *east*, *west*, *get*, *put*, and *charge*) in one action variable.

In this example, there are two different kinds of domain structure that a planner could exploit. First, a robot executing optimally will only ever want to go back and forth between three locations: the pickup, drop-off, and charger points, so the robot's view of the map can be abstracted into a smaller, topological version. Second, the battery sensor is more fine-grained than needed, so similar battery levels can be clustered together, say, in groups of 100 (see figure 4-1(b)).

Were the robot to attempt to find a policy for this domain using a single previous abstraction method such as state aggregation or temporally extended actions, it would miss the chance to exploit both types of structure. The module hierarchy framework, however, allows multiple abstraction methods to be used together, each focusing on the domain structure it is able to simplify.

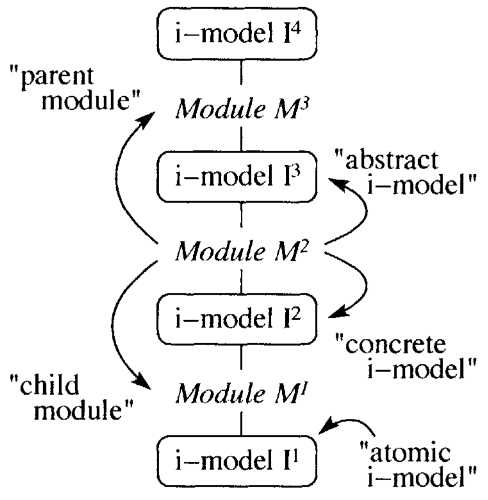


Figure 4-2: Module hierarchy nomenclature, with labels given relative to module M^2.

4.2 Module hierarchy

The module hierarchy begins with what the agent is given: a domain model expressed as a factored MDP and a reward function over the variables in the factored model.

In addition, the agent is supplied with a hierarchy of abstraction methods that dynamically create a hierarchy of abstracted versions of the base-level (given) domain model. In the framework, each abstraction method is packaged into an *abstraction module*, and individual modules are instantiated and combined in a *module hierarchy*, which is used to plan and act in a domain. The modules in the module hierarchy induce the successively more abstract versions of the base-level domain, called *i-models*. The module hierarchy thus ends up being an alternation of i-models and abstraction modules (see figure 4-2).

The top-level i-model is a trivial MDP with one state and one action (the action means "act in the domain"), while the bottom-level i-model, I^1, is identical to the input model.

4.2.1 Intermediate models

Given that the output of one module will be piped into another, and given that the two modules can be arbitrary and not specially designed to interface with each other, some intermediate language format is needed, some intermediate model that can store the results of one module output for feeding into another module. The modules listed in table 2.1 take, as an input format, an MDP, a semi-MDP, or a factored MDP. For all prior methods whose output is an abstract domain (rather than some execution system), the output format is one of the same three choices.

If it were necessary to feed a semi-MDP output into a module that takes factored MDPs as input or vice versa, there are three choices. The first and easiest choice is not to attempt the conversion, but it would be better to allow arbitrary abstraction modules to connect if at all feasible, since otherwise the module hierarchy may be severely limited in the kinds of domain structure it can deal with effectively.

The second choice is to convert the intermediate model into the correct format as needed. It is easy to convert from a factored MDP to a non-factored semi-MDP; all that needs to be done is to multiply out the factors and make all actions take one time-step. The reverse (non-factored semi-MDP to factored MDP) is also easy, because it just involves introducing time as a second state variable. But while these conversions are fairly easy, they hide or lose a lot of the model's structural information, and while it could theoretically be recovered, it is too time-consuming in practice to do so.

The final choice for making module inputs and outputs compatible is to have them be the same intermediate model format, and have that format be some generalization of both semi-MDPs and factored MDPs. This model type might be called a factored semi-MDP, giving it both the "factored" aspect to expose the domain structure and the "semi-" aspect to allow for temporal abstraction. Unfortunately, while semi-MDPs and factored MDPs both have fairly easy well-defined semantics, factored semi-MDPs are a different, more difficult case. The problem arises because of the interaction between the temporal aspect and the factored state and action spaces.

Factored semi-MDP issues One issue with factored semi-MDPs is how to represent the temporal extent of an action. Recall that each state variable has its dynamics expressed as a probability distribution over possible values and transition times. The probability distribution for temporal extent therefore has to be duplicated across the dynamics of all state variables affected by an action. Additionally, in order for the model to be consistent, each dynamics function must agree on the probability distribution for how long each action might take.

This redundant storage of temporal information causes extra dependencies. Given a particular action, a state variable might stay the same value no matter how long the action takes, but even so, its probability distribution over values and times would have to depend on whichever state variables affect the time distribution. For example, suppose that a robot takes more time to charge the lower its battery level is, and suppose that the robot's location never changes while it is charging. The probability distribution for the robot's location must depend on the robot's battery level, because it needs to know how long to take not to change in order to be consistent with how long the battery state will take to charge. The way around this unwanted situation is to separate the probability distribution over action times from the probability distributions for state variables; the latter become conditional probability distributions, conditional on the time that the action takes.

A second issue with factored semi-MDPs is whether to allow a single action to cause different state variables to transition at different times. Allowing this would violate the intuitive meaning of choosing and applying an action, and it would cause so many other semantic problems that it is really a non-issue. All state variables whose dynamics depend on the same action variable must transition simultaneously. (At least, they must be modeled as such. Given a domain where actions cause unsynchronized state variable transitions, it is sometimes possible to split such an action in two and rework the domain dynamics to achieve roughly the same effect as unsynchronized transitions.)

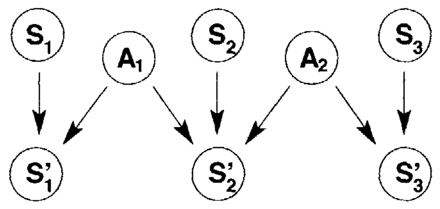


Figure 4-3: The transitions for states s_1 and s_3 are conditioned on synchronized actions.

A third issue with factored semi-MDPs is how to handle multiple action variables. Normally, multiple action variables would be thought of as being independent (as with multiple state variables). If two or more were to affect the dynamics of same state variable, however, then the reasonable thing to do seems to be to require the actions to terminate simultaneously. Otherwise, the same sort of unsolvable problems happen as with the case of multiple state variables transitioning separately when affected by the same action variable. While this solution takes care of the problem, it causes undesirable dependencies, as in figure 4-3, where the transition probability distributions for states s_1 and s_3 need to be synchronized at all times. An area of future research will be to come up with better representations for the probability distributions involved in specifying the factored semi-MDP. The goal will be to specify more explicitly whatever independence exists and take advantage of situations where, for instance, two actions affect the same variable in only some but not all situations.

A final issue with factored semi-MDPs is how to express the reward function. In factored MDPs and in semi-MDPs, the reward function gives the instantaneous reward based on the current state and the current action. In factored semi-MDPs with multiple unsynchronized action variables, though, there will be no current state for the entire system at times when some action is in the middle of executing. This problem can be solved easily by forbidding reward functions from using the values of state variables influenced by unsynchronized actions, or perhaps making additive reward functions, with one part for each synchronized part. If this solution is not possible because the reward really does depend on multiple parts, then the reward function might be able to use the last known value for those state variables that are in the middle of transitioning.

Since a lot of these issues only occur when there are multiple simultaneous action variables, the i-models are constrained so that only one action variable can be executing at any time. The representation and semantics for factored semi-MDPs with multiple unsynchronized action variables is an area that is being investigated by others [71] and may be ready for inclusion in the module hierarchy framework in the future.

i-models Each i-model I^j is an intermediate representation for the domain and is created by module M^{j-1} looking at I^{j-1} and applying some abstraction. The i-models are *factored semi-MDPs* as described above, since they need to be able to represent both the input model, which is a factored MDP, and also any temporal abstraction information generated by abstraction modules. These i-models may have multiple action variables, but only one action variable may be executing at any time.

Formally, an i-model I is defined as a tuple $(\overline{S}, \overline{A}, \tau, T, r, \gamma)$:

- A set of state variables $\overline{S} = \{S_1, \ldots, S_N\}$ where each S_i is a state variable taking values $\{s_{i1}, \ldots, s_{in_i}\}$; the state space as a whole is given by $S_1 \times \ldots \times S_N$.

- A set of action variables $\overline{A} = \{A_1, \ldots, A_M\}$ where each A_i is an action variable taking values $\{a_{i1}, \ldots, a_{im_i}\}$; the action space as a whole is given by $A_1 \cup \ldots \cup A_M$.

- A time probability distribution τ, where $\tau : \prod_{S \in \overline{S}} S \times \bigcup_{A \in \overline{A}} A \times \mathbb{N} \rightarrow \mathbb{R}$ gives the probability distribution over lengths of time that each action will take in each state.

- A set of component transition probability functions $T = \{t_1, \ldots, t_N\}$ where each t_i is a conditional probability function that gives the probability of states given the current state, the chosen action, and the action duration; the range of state and action variables for t_i are given by $\overline{S_i} \subseteq \overline{S}$ and $\overline{A_i} \subseteq \overline{A}$ respectively, and

$$t_i : \prod_{S \in \overline{S_i}} S \times \prod_{A \in \overline{A_i}} A \times \mathbb{N} \times S_i \rightarrow \mathbb{R}$$

; since t_i is a conditional probability distribution, for any state s, action a, and action duration n, $\sum_{s' \in S_i} t_i(s, a, n, s') = 1$.

- A reward function r, where $r : \prod_{S \in \overline{S}} S \times \bigcup_{A \in \overline{A}} A \rightarrow \mathbb{R}$.

- A discount factor γ, where $0 < \gamma \leq 1$.

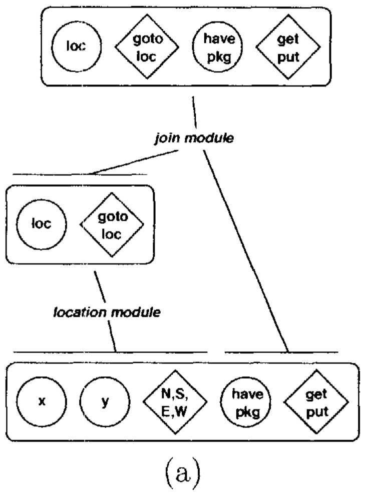
the action values in different action variables A are assumed to be unique. Therefore, $\bigcup_{A \in \overline{A}} A$ gives the set of all possible action choices (remember, only one action variable is active at any one time).

In each i-model, the distributions τ and $\{t_k\}$ as well as the reward function r are represented as algebraic decision diagrams (see section 5.2.3). This allows for a much more compact representation than, say, using tables or even using normal decision trees.

4.2.2 Linear vs. branching hierarchy

A linear hierarchy like the one in figure 4-2 is not the only way one could imagine modules successively abstracting different parts of an MDP. It is clear that the modules need to operate in an acyclic manner, and there should be one input model and one most-abstract model at the top (see section 4.4 for execution details on why).

One could imagine some sort of system, though, where abstraction modules did not operate on the whole domain but rather just one small part. In such a system, several such modules might operate in parallel and then have the results of their abstraction joined back together. Such a module hierarchy would have perhaps some sort of directed acyclic graph (DAG) structure (see figure 4-4 for a comparison).



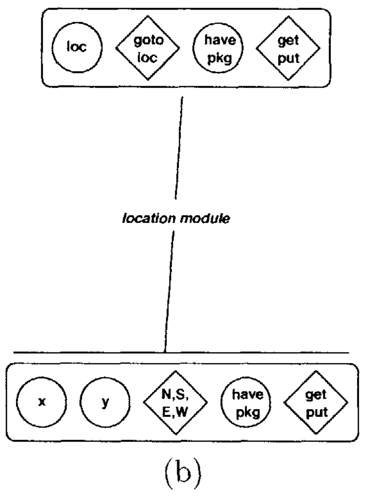


Figure 4-4: (a) Abstract the domain parts and merge them as needed with a join module. (b) Always abstract the whole model, even if changing only one part.

Such a DAG module hierarchy would be attractive because it would allow abstraction modules to operate in parallel; this would probably mean less work when changing the representation and re-abstracting, since the changes would have fewer abstraction modules to propagate through (see section 4.4.4). It is also attractive because it means that a single abstraction module does not have to consider the entire domain when it only wants to operate on a small part of it; i.e., it wouldn't have to copy an only slightly modified version of the entire domain from its concrete i-model to its abstract i-model.

The major problem with such a DAG module hierarchy is how to hook the modules together. It is unclear what format the output of a module (and the input of the next module) should be. Clearly, it has to be some sort of MDP fragment, perhaps some state and action variables along with transition and reward functions for them. Subtle problems arise, though, due to these MDP fragments not containing dynamics for the whole domain. For instance, suppose that some module creates a new action or a whole new action variable. What are the effects of that action variable on the states outside of that MDP fragment? This gets especially tricky when the states outside of that MDP fragment are themselves being abstracted or otherwise changed. One possible rule might be that an abstraction module has to operate on domain fragments that contain all state variables affected by the contained action variables, but, given that action variables tend to have wide-spread effects, this ends up making the module hierarchy linear, as formulated above.

In summary, while a DAG module hierarchy promises some nice benefits, it causes intractable problems and will not be pursued further. The linear module hierarchy does put more burden on the individual abstraction modules, forcing them to update transition and reward dynamics for state and action variables not of interest and not being directly operated on. This burden does require slightly modifying some previous abstraction methods in order to package them into modules, but it ensures that the i-models are consistent and well-formed.

4.3 Abstraction module interface

The other element of the module hierarchy aside from i-models is the abstraction modules. Thankfully, it isn't necessary to come up with a lot of new abstraction types to fit into this framework—all existing abstraction and approximation methods for MDPs described in the section on previous methods will work with it. The only caveat is that certain changes need to be made to these prior abstractions in order to make them fit the interface that the framework requires. Usually, this involves making sure that they operate smoothly on just part of a model, and making sure that they re-abstract properly when changes happen.

The abstraction modules M^j are the heart of the module hierarchy. Each module must conform to the following interface:

- Each module M^j must create an abstract i-model I^{j+1} from i-model I^j. I^{j+1} should contain approximately the same information as I^j, except that some structure or redundancy will have been factored out in an attempt to make the domain simpler.

M^j must be able to create I^{j+1} so that a specified atomic state is representable in I^{j+1}, given that it is representable in I^j. (See section 4.4.3 for details on representability.) During initial planning, this will be the starting state, but it may change over the course of execution.

- Each module M^j must respond to requests for re-abstraction, so that I^{j+1} changes appropriately when I^j changes. Ideally, the new I^{j+1} will be only slightly different than the old one, and this allows modules to reuse a lot of the information from the old I^{j+1} when computing the new one.

  The actual approach that each module takes to re-abstraction needs to be lazy; that is, it should create I^{j+1} not when notified that I^j has changed, but rather when the new I^{j+1} is first needed by module M^{j+1}.

- Each module M^j must be able to translate a state from its concrete i-model I^j to its abstract i-model I^{j+1}. There will not always be an abstract state corresponding to every concrete state, but there are only a couple of times where this translation is necessary, and the planning and execution framework guarantees that this translation is only requested when it is possible (see section 4.4.3).

  An abstraction module is not required to be able to do the reverse (map abstract to concrete states), because there may be multiple concrete states that correspond to a single abstract state.

- Each module M^j must provide the proper hooks for the execution framework, which involves expanding actions from I^{j+1} into actions in I^j and storing intermediate execution information. The interface for execution is approximately given by the following pseudocode functions:

  ```
  void    setAbstractAction(Action a);
  void    makeAtomicObservation(State s);
  boolean isExecutingAbstractAction();
  ```

```
Action  getNextAtomicAction();
```

These four functions are used by higher level modules to communicate with lower level modules during the course of execution. Briefly,

- The `setAbstractAction` function allows module M^i to execute an action exported by module M^{i-1}.
- The `makeAtomicObservation` function allows module M^i to notify module M^{i-1} of the current state.
- The `isExecutingAbstractAction` function allows module M^i to ask module M^{i-1} if, based on the current state, it has finished executing the action previously requested by M^i.
- The `getNextAtomicAction` function allows module M^i to ask module M^{i-1} for the next lowest level action corresponding to the abstract action that is currently executing in M^{i-1}.

Full details on execution are given in section 4.4.2.

All existing MDP abstraction and approximation methods that are described in this thesis can be made to fit this interface.

To solve the gridworld example domain, two existing abstraction methods are packaged into abstraction modules. The first module, a *subgoal-options* module, abstracts the robot's x-y coordinates into the three pertinent locations and abstracts the normal compass-direction actions into actions that go between the pertinent locations. The second module clusters together states that differ only by having slightly different similar battery levels. These modules are described in detail below in sections 5.1.1 and 5.1.2.

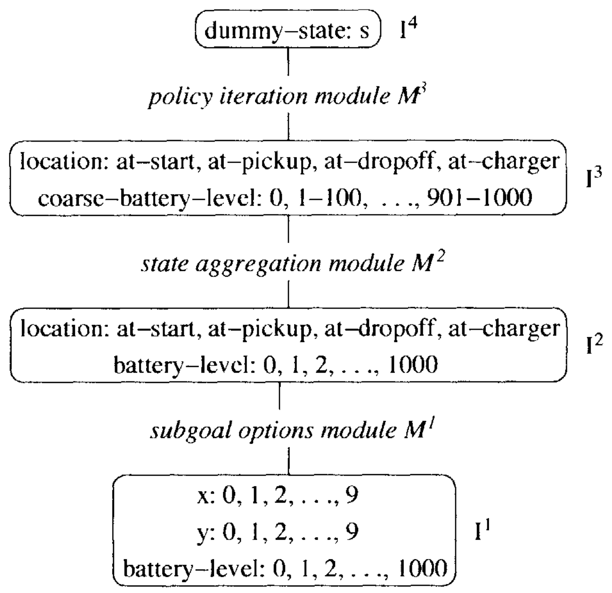

dummy–state: s
I^4
policy iteration module M^3
location: at–start, at–pickup, at–dropoff, at–charger
coarse–battery–level: 0, 1–100, . . ., 901–1000
I^3
state aggregation module M^2
location: at–start, at–pickup, at–dropoff, at–charger
battery–level: 0, 1, 2, . . ., 1000
I^2
subgoal options module M^1
x: 0, 1, 2, . . ., 9
y: 0, 1, 2, . . ., 9
battery–level: 0, 1, 2, . . ., 1000
I^1

Figure 4-5: A module hierarchy for the gridworld example.

4.4 Planning and execution

Given a module hierarchy that a domain expert has created, the module hierarchy framework creates various plan or policy pieces and then executes actions while monitoring its execution.

4.4.1 Planning

A module hierarchy for the example gridworld domain is given in figure 4-5. In the figure's i-models, the state variables are listed to demonstrate how the input model is successively abstracted.

The topmost module is always a special module that creates an abstract i-model with only one state and one action. This "abstraction" involves solving the model using policy iteration [67], so that the single abstract action is a temporally abstract action meaning "execute the optimal policy that was calculated using policy iteration."

Planning is implicit in the module hierarchy as part of the process of creating the i-models. The i-models are created in order from the bottom to the top, where each module M^j creates I^{j+1} from I^j. Some obvious planning is done by module M^3 as it uses policy iteration to solve the abstract model I^3, but implicit planning also occurs in the subgoal-options module M^1, because creating each option requires creating a plan or policy to get from one location to another. In most module hierarchies, the majority of the planning will happen this latter way, i.e., as part of the process of creating an abstract i-model, rather than in the topmost module.

The overall plan is therefore composed of pieces that are created by and stored (or even generated on the fly as needed) in the individual modules. Notice that each piece of planning is done in a much smaller domain than the whole 200,000 state domain: the subgoal-options module creates options on a 1100 state domain, and the policy iteration module creates a policy in an 88 state domain.

An important thing to note is that I^{j+1} is created so that a specified state is representable. When doing initial planning, the state specified to be representable is the starting state. This lets the subgoal-options module, for instance, know that it will have to create a location other than the pickup, drop-off, and charger locations if the initial state is elsewhere.

Another thing to note is that the planning is done in a lazy fashion, where i-model I^{j+1} is created only when it is first needed. This will be important to prevent unnecessary plan change propagation during re-planning.

Thus, when receiving a message to create i-model I^{j+1}, each module M^j follows these steps:

1. Pass the message about representing the world dynamics for a particular state to the child module (if any).

2. Note the particular state to represent.

3. Mark that I^{j+1} is out of date.

When receiving a request for I^{j+1}, each module M^j follows these steps:

1. If I^{j+1} is up to date, return it.

2. Ask M^{j-1} for I^{j}. If there is no child module, then this module's concrete i-model I^{j} is simply the input model.

3. Based on I^{j}, figure out what I^{j+1} should be. What this step actually does is specific to the type of abstraction being performed by M^{j}.

4. Mark that I^{j+1} is up to date.

5. Return the newly created I^{j+1}.

Since both of these lists of steps begin by asking something of the child module, a request to model a certain starting state or a request for the top i-model will result in a chain of messages being passed from the top to the bottom of the module hierarchy, and then the abstraction work is done in order from the bottom up to the top.

As can be seen from the above steps, the abstraction modules take a lazy approach to creating and updating world dynamics. This prevents several successive changes from propagating all the way up the module hierarchy, causing information to be calculated and then immediately replaced without being used. In the lazy approach, the modules store whatever information they need to about the abstract i-model that should be created, and they delay actually creating the abstract i-model until its information is requested.

4.4.2 Execution

After the i-models (and the relevant plan pieces) have been created, the module hierarchy begins to execute. This starts by executing the single action in the top i-model.

When an action is executed in any I^{j+1}, module M^{j} makes observations and chooses concrete actions in I^{j} to execute until the abstract action from I^{j+1} is done executing. Each time M^{j} executes an action in I^{j}, module M^{j-1} makes observations and chooses concrete actions, etc. The actions are translated further and further down the module hierarchy, and eventually they end up in I^1 as atomic actions that can be executed directly in original domain model. Each abstract action executes to completion, as determined by whichever module created it (except that all actions are interrupted when re-planning; see section 4.4.4).

This description makes it sound like the system expands the policy into a sequence of actions before executing any of them, but at the same time as the distributed policy is executing, the system is receiving observations about the current state. These are passed to the modules in the hierarchy, so that the distributed policy can be closed-loop.

The execution loop consists of two steps: informing all modules of the current state, and then asking the topmost module for the next atomic action to execute. When receiving a request for the next atomic action to execute, each module M^j follows these steps:

1. From M^{j-1}, get the next atomic action to execute.
2. If M^{j-1} returns a `terminated` action,
   (a) Choose the next action in I^j to execute, according to the current action that is executing in I^{j+1}.
   (b) If there is no such action, then the action in I^{j+1} is finished, so return a `terminated` action to M^{j+1}.
   (c) Tell M^{j-1} to execute the action chosen in I^j.
   (d) From M^{j-1}, get the next atomic action to execute.
3. Return the atomic action specified by module M^{j-1}.

From these steps, it can be seen that each module stores the currently executing action in its abstract i-model as part of the current system state. This recording of the currently executing action is the full extent of state update that the module hierarchy does.

Suppose the module hierarchy given in figure 4-5 is used to solve the example gridworld domain, starting as shown in figure 4-1(a). When the single top action is executed, M^3 gains control and executes the optimal policy that it has found (i.e., the top-level action never terminates). The *goto-pickup* action is executed, and control passes to M^2. M^2 passes the action on to M^1, which determines that the current concrete action corresponding to *goto-pickup* is *north*. This is in I^1 and atomic, so the robot takes this action. Suppose this action fails to move the robot; M^1 determines that *north* is again what should be done. This time, the action succeeds, and M^1 determines that *goto-pickup* has terminated. It therefore returns control to M^2, telling it that *goto-pickup* has terminated, and M^2 similarly returns control to M^3. Since *location* is now *at-pickup*, the optimal policy indicates that *pickup* should be executed. Execution continues in a similar manner.

4.4.3 State representation

When the module hierarchy is executing, it is not always the case that every i-model can represent the current state. For instance, some abstract actions may be temporally extended and in the middle of executing. But note that state update and state representation is only important for selecting actions, which means that the module hierarchy only needs to update each i-model's state when the current action for that i-model ends and a new action needs to be chosen.

This is good because it means that it is not necessary to continue to tell the module hierarchy to change in order to represent the dynamics for the current state; it only has to represent the dynamics at the initial state.

Of course, if someone asks what the current state is in some i-model that is in the middle of executing an abstract action, then there is no good answer. But there doesn't need to be, as long as the correct sequence of concrete actions keeps being generated, and indeed the correct actions do get generated. The transition dynamics of the model were correct (by assumption) when the plan was formulated. Those dynamics will still be correct, and therefore the plan will still be valid. It is just no longer the case that every i-model can model the current state.

Suppose that, as part of execution, a module M^j needs to determine the current state in I^j when choosing a new action. The current state is determined by having each module successively translate the observed (atomic) state up from I^1.

Not all atomic states are always representable at all i-models; for instance, only four of the hundred combinations of x and y correspond to values of the *location* state variable. This is not a problem, though, because the only time that the current state needs to be representable in I^j is when a new action is being selected in I^j, and this happens in only two situations. The first is when beginning to execute, and recall that the initial i-models are built so that the initial state is representable. The second is when some action in I^j has just finished, and the current state will necessarily be representable since I^j is assumed to be a valid factored semi-MDP.

A state can therefore be generally represented as a combination of the most recent state at each i-model, plus the length of time (if any) that each i-model's action has been executing. This state representation does not cause problems, because the only points at which the current state needs to be represented are when the next action must be chosen, and the module hierarchy is constructed so that it is exactly those points that are representable.

4.4.4 Re-planning and dynamic representation changes

The module hierarchy so far is a static entity: the decomposition and the modules are chosen, then the framework executes. What makes the module hierarchy different from similar previous methods is that the module hierarchy can change the representation dynamically and update the plan accordingly.

Each module must respond to requests for re-abstraction. These requests come about from several different sources. One source could be execution monitoring, described below, where a module realizes that the abstraction it has created doesn't adequately represent how the world is acting; the module re-abstracts in order to allow higher modules to know about the abstraction breaking down. Another source of changes is other modules, which may decide that the world's current state might be best represented differently.

In the gridworld example, the robot can make several deliveries on one charge, and so the robot's battery level isn't important until it gets low. Suppose a new module is inserted right below the policy iteration module and have it selectively remove or not remove the *coarse-battery-level* state from the abstract i-model that it creates (see figure 4-6(b)), say, removing *coarse-battery-level* when its value is above *1-100.* Removing *coarse-battery-level* gives the policy iteration module a much smaller model to find a policy for.

When the battery level gets low enough, the selective removal module should notice this and change I^4 to include *coarse-battery-level*, causing the policy iteration module to update the optimal policy it has found to take account of the new *coarse-battery-level* state variable.

In general, when a module M^j changes I^{j+1}, this will cause a cascade of updates up the hierarchy as each module propagates the change by updating its abstract i-model. These updates happen in the same way that initial planning happened: modules are notified that their abstract i-models are out of date, modules are told to create i-models so that the current state is representable, and when the new i-models are requested, each I^{j+1} is created based on I^j.

It is desirable to be able to re-plan at any atomic state, but it is likely that a request for re-planning will occur when abstract actions in several I^j are in the middle of executing. This is why all new i-models must ensure that the current state is representable when adjusting to changes. Since abstract actions may not be optimal or may not even exist any more, all currently executing actions are terminated before

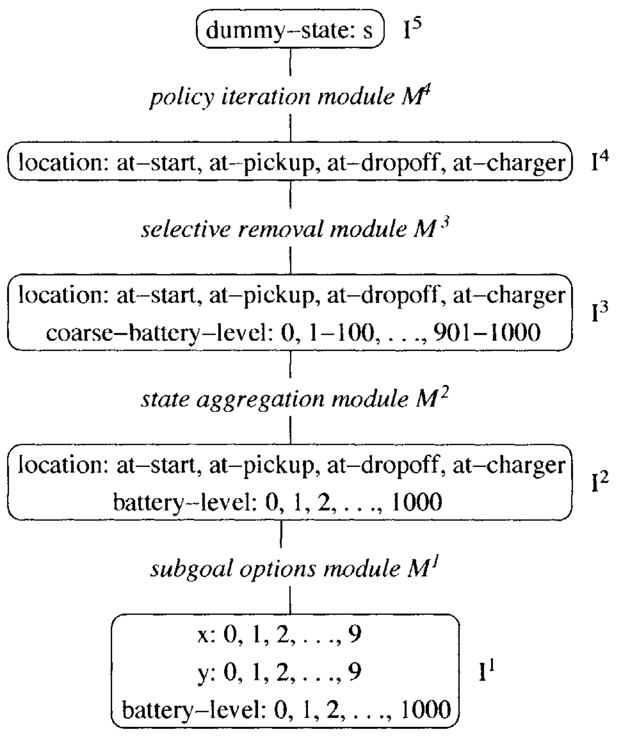

dummy–state: s
I5
policy iteration module M4
location: at–start, at–pickup, at–dropoff, at–charger
I4
selective removal module M3
location: at–start, at–pickup, at–dropoff, at–charger
coarse–battery–level: 0, 1–100, . . ., 901–1000
I3
state aggregation module M2
location: at–start, at–pickup, at–dropoff, at–charger
battery–level: 0, 1, 2, . . ., 1000
I2
subgoal options module M1
x: 0, 1, 2, . . ., 9
y: 0, 1, 2, . . ., 9
battery–level: 0, 1, 2, . . ., 1000
I1

Figure 4-6: The same module hierarchy as in figure 4-5 but with a selective removal module added.

the re-representation takes place. To continue execution after re-planning, the top i-model's single action must be executed again, just like the initial execution step.

Note that changes do not have to be propagated down the hierarchy towards I^1, but rather only from the point of the change up to the top of the hierarchy. It could be that the new requirement that the i-models be able to represent the current state is not met by some lower i-models, if they were in the middle of executing a temporally abstract action that was terminated. To ensure that this does not happen, each module M^j may only make changes to I^{j+1} when an action has just finished executing in I^j.

It would be possible to allow a module M^j to make changes to I^{j+1} in the middle of an action executing in I^j; this would necessitate halting the currently executing actions in lower i-models and asking modules to make the current state representable. It is unclear, though, that this would be beneficial. Module M^j decides how to construct I^{j+1} given the model it sees in I^j. If an abstract action in I^j is continuing to execute, why would module M^j suddenly decide that the representation of I^{j+1} should be changed? It seems that only new information about I^j, i.e., a cascading representation change or a new state, should cause M^j to ponder updating I^{j+1}.

As each module M^j updates I^{j+1} to take account of changes, it could recalculate I^{j+1} from scratch, but in many cases, it can reuse most of the solution from the previous version of I^{j+1}. This causes the change of representation to happen much faster than the creation of the initial representation.

For instance, suppose the robot is periodically instructed to change its drop-off location to one of nine other possible sites. Instead of representing all possible drop-off locations and having nine of them be useless, the drop-off location can be approximated as fixed in one place. When its location changes, the options module only has to create a new *goto-dropoff* option; it can reuse the options to reach other locations. Even better, changes to *location* cause the state aggregation module no new work, since it just copies information about *location* from I^2 to I^3. This representation strategy gives up the ability to model and plan for changes in the drop-off location, but it gains an order of magnitude decrease in state space size. For a large domain, tractability depends on making such tradeoffs, giving up some optimality in exchange for domain simplification.

Re-abstraction done carefully can therefore occur fairly frequently in this framework without being a burden. For large domains, this ability to keep the current representation small will likely mean the difference between tractability and intractability.

The ability to adapt the representation dynamically can be used in other ways as well. For instance, if more processing power is suddenly available, it may be advantageous to reduce the amount of approximating, in the hopes of getting a better policy. Or, if a better atomic model of the domain's dynamics becomes available (say, because it is being learned online), then that better model can replace the old model without needing to plan from scratch.

Sensitivity analysis When transition probabilities or rewards change and there is some current policy being executed, then that policy's value may change, and there also may be a new policy that the module hierarchy would calculate to be best. If the old best policy is not expected to gain reward that is very different than the reward a new best policy would gain, then it would likely be to the module hierarchy's advantage not to re-plan. Any changes in dynamics and reward that are determined actually to be significant would still be propagated up, so that more abstract levels can adjust their dynamics accordingly. Since dynamics changes may be frequent, this selective suppression of insignificant changes allows re-planning effort to be spent only on changes that really make a difference. The effort saved will not only be for a module that decides not to re-plan, but also for all modules above it in the hierarchy that subsequently have no cascading changes to adapt to.

In order to figure out whether a dynamics change is important, a module needs to combine information about the type and magnitude of the change with information about the type of abstraction it creates, performing some sort of sensitivity analysis. Much research has been done on sensitivity analysis in linear programs (which is what the optimal policy problems are), but it is currently unclear how to apply linear program sensitivity analysis results (or any other results) to the task of figuring out how much expected reward is lost. This is because a module would need to know not just how some dynamics change affected itself, but also how that dynamics change affected all modules above it in the hierarchy. In certain cases, it is conceivable that some seemingly insignificant change in dynamics at a lower level of the hierarchy results in a radical change in higher levels. Also, multiple modules could suppress dynamics changes that are insignificant separately but significant when combined.

At this time, therefore, modules always propagate changes and never suppress them. How to incorporate sensitivity analysis so as to avoid unnecessary re-planning is an area of on-going research.

Execution monitoring There are two types of execution monitoring that the module hierarchy could conceivably do. In one type, each module monitors its concrete i-model for large changes that would affect the abstraction it has produced for its abstract i-model. For instance, a robot might normally ignore its battery when it is charged enough, but when the battery gets low, suddenly the movement dynamics might include the information that a dead battery means no movement. Alternately, if a robot unexpectedly sees a pile of money in the distance, it may re-plan its movement to drive by the money and pick it up. Such large changes in the dynamics or reward need to be propagated up if they might affect policies' optimality at higher levels.

The second type of execution monitoring is where some module is in the middle of executing an abstract action, and it monitors the actual execution of the abstract action in order to compare with the dynamics that it exported to the abstract i-model. If the actual and expected dynamics differ significantly, then the difference is assumed to be due to a low probability occurrence and not due to the model being wrong. (If the model could be wrong, then the information about the difference could be used to learn the input domain model a little better, and then the changes could be propagated through the module hierarchy. It is less clear how such learning could occur in the middle of the module hierarchy and the information be pushed down to the base model.) In any case, the agent may be stuck in the middle of an inappropriate abstract action and wish to reconsider its choice of action in that model.

These types of execution monitoring can happen throughout the module hierarchy, even in the middle of abstract action execution, where it doesn't happen with normal semi-MDPs. The reason is that a normal semi-MDP doesn't have any way to stop in the middle of a temporally extended action and reconsider that action. There is not necessarily a state representation for being part-way through that action. In the module hierarchy, on the other hand, there is always lowest-level state corresponding to the current state, and every module can be told to re-abstract in order to represent this current state. It makes sense to take advantage of this ability to interrupt temporally extended actions that are having low-probability effects or when a new representation suddenly becomes more desirable.

There are several possible update policies when monitoring the execution and expected versus actual dynamics. One updates every i-model after every change, propagating any changes up the hierarchy to the top. The other updates the changed low level i-model but only propagates changes up to parent i-models if they are significant. It would be better to do the latter when possible, but, as discussed above, it is not yet possible to apply sensitivity analysis to separate significant changes from insignificant ones.

4.4.5 Advantages of planning/execution system

One advantage of using this planning and execution system is that re-planning can occur at any time, and when it does occur, it only occurs at the necessary level(s) of abstraction. That is, if a module needs to re-plan, but the lower level modules still validly represent their sub-domain's dynamics, then those lower levels don't have to re-plan.

This ties in with the advantage that the plans created are partial; that is, the agent doesn't plan in excruciating detail its next $10,000$ time-steps, but rather its plans are more vague (i.e., abstract) the further they are in the future. This isn't something that has to be carefully arranged, but it is a product of using the module hierarchy. For instance, suppose an agent is given several successive tasks to do, where each one involves using some state variables that are particular to that task. While executing each task, the agent can partially or completely ignore the state variables belonging to other tasks. It only needs enough knowledge of other tasks' state variables to know that the tasks are solvable and that its overall plan will work. As the agent addresses each task, that task's state variables will be represented at a more detailed level, allowing for a detailed local plan.

This multi-resolution approach means that near-future parts of the plan can change without needing to re-plan far-future parts of the plan. For instance, in the same example, if a future task's dynamics change, then the agent will have little or no wasted effort because it has only thought about the task in enough detail to know that it is doable.

Another advantage (which is essentially of an advantage of using the module hierarchy to model the whole domain) is that the planning problem is carved up into separate subproblems, each of which can be solved separately. Since planners usually take time at least polynomial in the size of the domain, this division of the planning problem is good for efficiency.

The module hierarchy is also good for efficiency because the plans or other calculations created in the i-models can be reused. Each i-model applies to possibly many different situations in the world; therefore, each policy that there is space to cache can be reused for each and every situation in the world that is "equivalent" as far as the i-model's perspective is concerned.

4.5 Optimality

In order to deal with very large domains, the module hierarchy gives up hope of hard optimality (or at least of provable hard optimality). There are times, though, when it would be nice to have at least an estimate of how much expected reward will be lost for a particular module hierarchy. This turns out to be a difficult problem because it is so dependent on the types of modules and their arrangement in a hierarchy, and because individual modules make different sorts of optimality claims if they make them at all.

Certain optimality bounds are immediately apparent. For instance, if each module in the hierarchy guarantees that a solution to its abstract i-model is as good as a solution to its concrete i-model, then the overall module hierarchy can guarantee hard optimality. Model minimization and the options framework, for instance, are two abstraction methods that make this guarantee. However, it turns out to be difficult to go beyond this.

To make this discussion concrete, consider the small module hierarchy as given in figure 4-7. The i-models are I^1, I^2, and I^3, listed from most concrete to most abstract, and the abstraction modules are M^1 and M^2. Let π_i^* be the optimal policy for i-model I^i, and let $V_i^\pi(s)$ be the long-term expected value of starting at state s in i-model I^i and taking actions according to policy π.

Suppose that each module in the hierarchy gives a bound on expected reward loss. The best possible scenario would be where it is possible to sum these bounds to find the overall expected reward loss of the module hierarchy, but it is not that simple. Normally, if an abstraction module M^2 gives an expected reward loss bound, it will

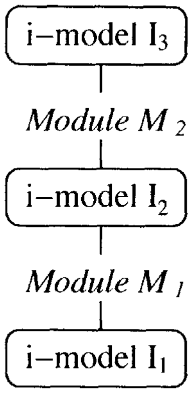


Figure 4-7: An example module hierarchy for the optimality discussion.

give the following difference:

$$\left|V_1^{\pi_1^*}(s) - V_1^{\pi_2^*}(s)\right|.$$

(This is a slight abuse of notation, because I^2 and I^1 may not have the same state and action spaces, and so the policy π_2^* may not be executable in I^1. The meaning of $V_1^{\pi_2^*}(s)$ is the long-term expected value of starting in state s and following a policy in I^1 that is π_2^* as translated during execution by module M^2.)

This difference only gives information about the loss incurred when executing I^2's optimal policy in I^1. In particular, it does not say anything about the loss incurred when executing some policy that is only *approximately* optimal for I^2. Unfortunately, this is exactly the case when calculating the estimated reward loss for the whole module hierarchy. In particular, the following difference is what is needed:

$$\left|V_1^{\pi_1^*}(s) - V_1^{\pi_3^*}(s)\right|.$$

It is unclear exactly what relation π_3^* will have to π_2^*, and so it seems that normal estimates of expected reward loss cannot be combined for successive abstractions in a meaningful way.

A different and actually successful approach involves value functions. The triangle inequality gives the following bound:

$$\left|V_i^{\pi_i^*}(s) - V_i^{\pi}(s)\right| \leq \left|V_i^{\pi_i^*}(s) - V_{i+1}^{\pi_{i+1}^*}(s)\right| + \left|V_{i+1}^{\pi_{i+1}^*}(s) - V_{i+1}^{\pi}(s)\right| + \left|V_{i+1}^{\pi}(s) - V_i^{\pi}(s)\right|.$$

In words, this says that the difference in expected value at some state between the following optimal policy and following some other policy is bounded by the sum of three terms: first, the difference in value between following an optimal policy in the original domain and following an optimal policy in the abstract domain; second, the difference in value between following an optimal policy in the abstract domain and following the other policy in the abstract domain; and third, the difference in value between following the other policy in the abstract domain and following the other policy in the original domain.

This inequality can be used to expand the middle term on the right-hand side into three terms involving value functions for i-models I^{i+1} and I^{i+2}. Applied to the example module hierarchy, this gives

$$\begin{aligned}\left|V_1^{\pi_1^*}(s) - V_1^{\pi_3^*}(s)\right| \quad \leq \quad & \left|V_1^{\pi_1^*}(s) - V_2^{\pi_2^*}(s)\right| + \\ & \left|V_2^{\pi_2^*}(s) - V_3^{\pi_3^*}(s)\right| + \underbrace{\left|V_3^{\pi_3^*}(s) - V_3^{\pi_3^*}(s)\right|}_{=0} + \left|V_3^{\pi_3^*}(s) - V_2^{\pi_3^*}(s)\right| + \\ & \left|V_2^{\pi_3^*}(s) - V_1^{\pi_3^*}(s)\right|\end{aligned}$$

The central term is always zero since the optimal policy is always chosen in the topmost i-model. For a module hierarchy with k i-models, this equation generalizes to

$$\left|V_1^{\pi_1^*}(s) - V_1^{\pi_k^*}(s)\right| \leq \sum_{i=1}^{k-1} \left|V_i^{\pi_i^*}(s) - V_{i+1}^{\pi_{i+1}^*}(s)\right| + \sum_{i=1}^{k-1} \left|V_{i+1}^{\pi_k^*}(s) - V_i^{\pi_k^*}(s)\right|.$$

Munos and Moore [53] give some techniques for calculating the absolute values on the right-hand side of the equation.

Unfortunately, the above bound is rather loose, but loose loss bounds such as one above are all that can be derived in general, if the system allows modules powerful enough to drastically reduce the complexity of the original domain.

Chapter 5

Abstraction modules

As discussed above, abstraction modules conform to a standard interface, allowing the modules to be connected together in any way that makes sense for the domain to be solved. This section discusses a collection of modules created to explore the possibilities of the module hierarchy framework, as well as exploring some issues in the creation and parameterization of modules. The discussion of example applications of these modules references an example domain that is a simplified version of the computer game `nethack`. In this text-based adventure game, the player tries to maintain his health and avoid hunger while battling monsters in an attempt to escape from a multi-level dungeon; see section 6.1 for more details.

All abstraction methods (i.e., not MDP-solving methods) mentioned in the previous work section can be packaged into an abstraction module.

5.1 Modules needing parameterization

5.1.1 Subgoal-options module

The first abstraction discussed in the gridworld example above, having abstract actions that move the robot between the pertinent locations, is similar to the options framework [65] and to nearly deterministic abstractions [45]. The idea of the options framework is to create temporally extended actions, in order to speed up value/policy iteration, or in order to create a temporally abstract model that skips past most states by only executing the options (rather than the atomic actions that the options utilize). In this module, the options that it creates are all sub-policies to go from one salient location to another. Applied to the gridworld example, the resulting abstract i-model is a semi-MDP that has 3 locations instead of 100 x-y combinations.

The inputs to a subgoal-options module M^j are

- S_G, a set of goal states, where each goal state $\sigma \in S_G$ specifies values over some subset $\overline{S_{\text{goal}}} \subseteq \overline{S^j}$ of I^j's state variables; and
- $\{a_{\text{goal}}\} \subseteq A^*$, a set of action values, drawn from an action variable $A^* \in \overline{A^j}$ of I^j, that the options are permitted to use when attempting to reach a goal. These actions will be replaced by the options when the abstract model I^{j+1} is created.

In the example, $\overline{S_{\text{goal}}} = \{x, y\}$, $S_G = \{(2,2), (8,2), (5,9)\}$, and $\{a_{\text{goal}}\} = \{\mathit{north}, \mathit{south}, \mathit{east}, \mathit{west}\}$.

This module creates a set O of options, one for each goal $g \in S_G$. An option o_g gives a sub-policy that terminates when the restriction of the current state σ to $\overline{S_{\text{goal}}}$ is g. Each option is built by using policy iteration on a modified version of the domain, where the option's corresponding goal g has absorbing dynamics and a slightly positive pseudo-reward. The purpose of this fake reward is to entice an agent acting in the domain towards the goal without changing what the agent would do along the way, and so it is chosen to be several orders of magnitude smaller than any positive reward otherwise obtainable in the domain. For each option, the probabilistic expected time transition, state transition, and reward functions (*pett*, *pest*, and *per*, respectively) are calculated for moving from goal state to goal state.

The abstract state and action variable sets that this module creates for I^{j+1} are

- $\overline{S^{j+1}} = \overline{S^j} \setminus \overline{S_{\text{goal}}} \cup \{S_G\}$; and
- $\overline{A^{j+1}} = \overline{A^j} \setminus \{A^*\} \cup \{A'\}$, where A' is an action variable with values $\{a^*\} \setminus \{a_{\text{goal}}\} \cup O$.

In the gridworld example, the robot's x and y state variables are replaced with a state variable whose values are the three salient goal locations, and the robot's choices to move in the four compass directions are replaced by actions to move from one goal location to another.

Let $u : \prod_{S \in \overline{S^{j+1}}} S \rightarrow \prod_{S \in \overline{S^j}} S$ be a mapping that unpacks the goal part of a state in I^{j+1} into its constituent state variables, giving a state in I^j. In other words, u is the obvious mapping from S_G to $\overline{S_{\text{goal}}}$ extended to be the identity on other state variables. The subgoal-options module maps $I^j \rightarrow I^{j+1}$ and defines τ^{j+1}, T^{j+1}, and r^{j+1} in terms of their I^j counterparts as follows:

- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) = \begin{cases} pett(u(\sigma^{j+1}), \alpha, n) & \text{if } \alpha \in O \\ \tau^j(u(\sigma^{j+1}), \alpha, n) & \text{if } \alpha \notin O \end{cases}$

- $t_k^{j+1}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1})$
  $= \begin{cases} pest_k(u(\sigma^{j+1}), \alpha, n, u(\sigma'^{j+1})) & \text{if } \alpha \in O \\ t_k^j(u(\sigma^{j+1}), \alpha, n, u(\sigma'^{j+1})) & \text{if } \alpha \notin O \end{cases}$

- $r^{j+1}(\sigma^{j+1}, \alpha) = \begin{cases} per(u(\sigma^{j+1}), \alpha) & \text{if } \alpha \in O \\ r^j(u(\sigma^{j+1}), \alpha) & \text{if } \alpha \notin O \end{cases}$

5.1.2 State-aggregation module

The second abstraction discussed in the gridworld example above, clustering together states that have similar battery levels, is a simple state aggregation [4]. Using the mapping between original and abstract states, transition and reward dynamics for the new states can be formed by taking the average (mean) of the dynamics for the corresponding original states.

The inputs to a state-aggregation module M^j are

- $S_{\text{nonaggr}} \in \overline{S^j}$, the state variable being transformed;

- S_{aggr}, the replacement state variable; and

- $f : S_{\text{nonaggr}} \to S_{\text{aggr}}$, the aggregation function.

In the gridworld example, S_{nonaggr} = *battery-level*, S_{aggr} = *coarse-battery-level*, and $f(x) = \lceil x/100 \rceil$.

The abstract state and action variable sets that this module creates for I^{j+1} are

- $\overline{S^{j+1}} = \overline{S^j} \backslash \{S_{\text{nonaggr}}\} \cup \{S_{\text{aggr}}\}$
- $\overline{A^{j+1}} = \overline{A^j}$

For any state $\sigma^j \in \prod_{S \in \overline{S^j}} S$, let $f(\sigma^j)$ be the same state but with the value v of S_{nonaggr} in σ^j replaced by $f(v)$. Also, let $c(\sigma^{j+1})$ be the number of states that f maps to σ^{j+1}, i.e., $c(\sigma^{j+1}) = |\{\sigma^j : f(\sigma^j) = \sigma^{j+1}\}|$. The state aggregation module maps $I^j \to I^{j+1}$ as follows:

- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) = \frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} \tau^j(\sigma^j, \alpha, n)$
- $t_k^{j+1}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1}) =$
  $\frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} \sum_{\sigma'^j \in f^{-1}(\sigma'^{j+1})} t_k^j(\sigma^j, \alpha, n, \sigma'^j)$
- $r^{j+1}(\sigma^{j+1}, \alpha) = \frac{1}{c(\sigma^{j+1})} \sum_{\sigma^j \in f^{-1}(\sigma^{j+1})} r^j(\sigma^j, \alpha)$

This module takes the uniform average of the dynamics and reward over the states being aggregated. This is clearly an approximation, since the transition probability distribution at an aggregated state depends very much on the underlying distribution over the states that were aggregated, which in turn depends on the agent's actions to this point. Since the goal is to work in huge domains, the large reduction in state space size outweighs the risk of abstracting out pertinent information.

5.1.3 Ignore-state-variables module

The *ignore-state-variables* module selectively hides or reveals certain state variables and actions based on the current state. In the `nethack` example, this module was used to ignore the food aspect of the domain (the player's hunger level, whether the player was carrying food, and what food was on the ground) when the player was not hungry. This module was similarly used to ignore the health aspect of the domain as appropriate, and it was used to ignore items in the domain when they were no longer there (e.g., killed monsters or eaten food).

The idea behind this module is to take advantage of the locality of relevancy in MDPs; that is, certain parts of the state space are interesting at different times, and certain state space variables are interesting at different times. The module's input parameters are a list of state variables and action values possibly to ignore, along with a function that says when to ignore them.

This module might be classified as a sort of state aggregation module, since it clusters together states that have the same values over all state variables except the ignored ones. The abstract i-model created by this module is like the concrete i-model except that certain state variables and action values may be missing. If the transitions for some not-removed state variables condition on the values of removed state variables, then those inputs to the probability distribution are fixed at their current value. For instance, if health is currently 21 and is being ignored, then if the transition probabilities for the x coordinate depend on health, the abstract transition probability distribution is created by choosing the part of the concrete distribution where health is 21. The reward is similarly abstracted.

Currently, the abstract dynamics created by this module are not updated while the state and action items continue to be ignored; for instance, if health decreases from 21 to 20 but should still be ignored, then the abstract dynamics are not updated to reflect what happens when health is 20 rather than 21. This design choice stemmed from the desire to avoid unnecessary updates to the dynamics in the rest of the module hierarchy. Obviously, the module could be changed to see if the dynamics have changed or not when, say, health drops from 21 to 20, and the module could propagate a dynamics update if a change did occur.

Better yet (but still an area of future research) would be to apply sensitivity analysis to understand when changes are significant enough to warrant changed dynamics. TODO: better yet is to give an example of a sensitivity cliff that it's easy to wander off of, or maybe refer to the section below on automatically aggregating states.

5.1.4 Split-on-state and join-on-state modules

These two modules allow subparts of the state space to be solved separately and recombined, in a way similar to the macro-action framework [31].

These modules are not as simple as they first appear. The issue has to do with the model being a factored model, with state variables rather than atomic states. In the work of Hauskrecht *et al.* on regular MDPs, boundary states were added to each region's set of states, where the boundary states represented those states one step outside of a region. For instance, in the `nethack` example, each region would be composed of the states for a whole level plus one single state for each of the top of the ascending staircase and the bottom of the descending staircase. These single out-of-level states are the goals to be reached by macro-actions. In regular MDPs, it was easy to add the new goal states to the model; in a factored MDP, however, single new states cannot be added, only single state variables.

This would not be an issue if the split-on-state and join-on-state modules would always occur one after another in the module hierarchy. However, the idea behind splitting up the domain is to allow each part to be processed separately, by different modules in different ways. Each region's state variables therefore contain information about whatever boundary states exist. Also, the join-on-state module is able to handle each region of state space being abstracted differently, by requesting information from the split-on-state module and by utilizing specially created actions that cause the state to jump from one subpart to another.

Split-on-state module The split-on-state module divides up the state space into subparts. Each state is assigned to a subpart based on the state value that it contains for a particular *indicator* state variable. In the `nethack` domain, for instance, the dungeon was divided up based on the *floor* state variable, so that an escape path could be found separately through each floor. Copies of all the rest of the state variables are created for each subpart, as well as copies of all the action variables.

The transition dynamics for each subpart are simply the original dynamics but with the indicator state variable fixed, if the active action variable is from that subpart. If it the active action variable is from another subpart, then the state for this subpart does not change no matter what action is chosen. The abstract reward function is based on what happens in the active subpart only.

Since there will be other modules between the split-on-state and join-on-state modules that will want to abstract each subpart separately, it would be good to have each subpart be as independent as possible. Unfortunately, because some actions can cause the state to switch into other subparts, the subparts will have to be linked somehow (i.e., transition probability distributions for state variables in a subpart will be dependent on some state or action variables not in that subpart). This interdependence is made as small as possible by creating a new action variable A_{switch} to encapsulate all dynamics that switch subparts, removing such dynamics from each subpart's normal action variable.

A_{switch} contains action values for each combination of state and action that can cause the subpart to switch, and those action values execute the switching action when at states where the subpart could switch but execute a no-op elsewhere. For instance, in the `nethack` domain, the player can go from floor to floor using stairways. So, A_{switch} contains an action value meaning "go up this stairway" for each stairway; this action value causes the player to ascend when at the bottom of the stairway and to stay put elsewhere. In the split action variable corresponding to the floor at the bottom of the stairway, the corresponding dynamics (i.e., the result of taking an *up* action at the bottom of the stairway) are changed to staying put. By moving all switching behavior to A_{switch}, each subpart's dynamics are virtually independent, which is a huge boon for modules operating between the split-on-state and join-on-state modules.

The single input to a split-on-state module M^j is $S_{\text{split}} \in \overline{S^j}$, the state variable that indicates the current subpart. In the `nethack` domain, $S_{\text{split}} =$ *floor*.

The abstract state and action variable sets that the split-on-state module creates for I^{j+1} are

- $\overline{S^{j+1}} = \{S_{\text{split}}\} \cup \bigcup_{subpart \in S_{\text{split}}} \bigcup_{S \in \overline{S^j}} S_{subpart}$

- $\overline{A^{j+1}} = \{A_{\text{switch}}\} \cup \bigcup_{subpart \in S_{\text{split}}} \bigcup_{A \in \overline{A^j}} A_{subpart}$, where A_{switch} is as described above.

For any action value $\alpha^{j+1} \in \bigcup_{A \in \overline{A^{j+1}} \setminus A_{\text{switch}}} A$, let $sp(\alpha^{j+1}) \in S_{\text{split}}$ be the subpart that the action value is drawn from. For any state $\sigma^{j+1} \in \prod_{S \in \overline{S^{j+1}}} S$, let $\lfloor \sigma^{j+1} \rfloor$ be the corresponding state in I^j, where the subpart of σ^{j+1} that is mapped into $\overline{S^j}$ is given by the value of S_{split} in σ^{j+1}. Also let $\lfloor \sigma^{j+1} \rfloor_{s_{\text{sp}}}$ be the same except that the selected subpart of σ^{j+1} is given by $s_{\text{sp}} \in S_{\text{split}}$.

For action values $\alpha \in A_{\text{switch}}$, let $aps(\sigma^j, \alpha) \in \{\text{true}, \text{false}\}$ indicate whether σ^j is an appropriate pre-state for α, and let $u(\alpha)$ be the corresponding normal action value that is executed when α is at an appropriate pre-state.

The split-on-state module maps $I^j \rightarrow I^{j+1}$ as follows:

- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) =$
$$\begin{cases} \tau^j(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, u(\alpha), n) & \text{if } \alpha \in A_{\text{switch}} \\ \tau^j(\lfloor \sigma^{j+1} \rfloor_{sp(\alpha)}, \alpha, n) & \text{if } \alpha \notin A_{\text{switch}} \end{cases}$$

- For each subpart s_{sp}, $t^{j+1}_{k,s_{\text{sp}}}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1}) =$

$$\begin{cases} \text{if } \alpha \in A_{\text{switch}}: \\ \quad t_k^j(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, u(\alpha), n, \lfloor \sigma'^{j+1} \rfloor) \\ \qquad\qquad\qquad \text{if } aps(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, \alpha) \\ \quad 1.0 \quad \text{if } \neg aps(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, \alpha) \text{ and } \lfloor \sigma^{j+1} \rfloor = \lfloor \sigma'^{j+1} \rfloor \\ \quad 0.0 \quad \text{if } \neg aps(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, \alpha) \text{ and } \lfloor \sigma^{j+1} \rfloor \neq \lfloor \sigma'^{j+1} \rfloor \\ \text{else}: \\ \quad t_k^j(\lfloor \sigma^{j+1} \rfloor_{sp(\alpha)}, \alpha, n, \lfloor \sigma'^{j+1} \rfloor_{sp(\alpha)}) \quad \text{if } sp(\alpha) = s_{\text{sp}} \\ \quad 1.0 \quad \text{if } sp(\alpha) \neq s_{\text{sp}} \text{ and } \lfloor \sigma^j \rfloor_{S_k^j} = \lfloor \sigma^{j+1} \rfloor_{S_k^j} \\ \quad 0.0 \quad \text{if } sp(\alpha) \neq s_{\text{sp}} \text{ and } \lfloor \sigma^j \rfloor_{S_k^j} \neq \lfloor \sigma^{j+1} \rfloor_{S_k^j} \end{cases}$$

(Recall that t_k^j gives the post-state for the state variable S_k^j from I^j.)

- $r^{j+1}(\sigma^{j+1}, \alpha) =$
$$\begin{cases} r^j(\lfloor \sigma^{j+1} \rfloor_{sp(u(\alpha))}, u(\alpha)) & \text{if } \alpha \in A_{\text{switch}} \\ r^j(\lfloor \sigma^{j+1} \rfloor_{sp(\alpha)}, \alpha) & \text{if } \alpha \notin A_{\text{switch}} \end{cases}$$

Join-on-state module The join-on-state module merges back together the subparts that the split-on-state module created. As with the macro-action framework, the abstract states created by this module are the boundary states, where it is possible to go from one subpart to another, and the abstract actions are sub-policies to travel from one boundary state to another.

Even though the join-on-state module simply undoes the partitioning of the split-on-state module, it cannot get away with only using the same parameters as split-on-state module. This is because there may have been other modules, in between the split- and join-on-state modules, that reworked various parts of the state and action space until it is not recognizable as belonging to a particular subpart. So, the join-on-state module needs parameters that tell it what state and action variables correspond to which subpart.

The inputs to a join-on-state module M^j are

- S_{split}, the state variable that indicates the current subpart;

- A_{switch}, the action variable that switches subparts;

- $sp_s : \overline{S^j} \longrightarrow S_{\text{split}}$, a mapping that indicates which subpart each state variable belongs to (if any; S_{split} doesn't belong to a subpart); and

- $sp_a : \overline{A^j} \longrightarrow S_{\text{split}}$, a mapping that indicates which subpart each action variable belongs to.

These inputs are used in the construction of the abstract state and action spaces. The abstract state and action variable sets that the join-on-state module creates for I^{j+1} are

- $\overline{S^{j+1}} = \{S_{\text{bdry}}\}$, where S_{bdry} is a state variable whose values are all states that can be reached by taking an action in A_{switch} and attempting to go from one subpart to another. (More precisely, the state values are the restriction of such states to the post-subpart, along with the value of S_{split}.)

- $\overline{A^{j+1}} = \{A_{\text{gotobdry}}\}$, where A_{gotobdry} is an action variable whose values consist of all sub-policies a_{gotobdry} of the following form: given the current subpart and some $a_{\text{switch}} \in A_{\text{switch}}$ that switches from this subpart to another, attempt to go and execute a_{switch} in an optimal way. These policies are built in the same was as the options in the subgoal-options module, by running policy iteration on a domain with a slightly positive reward at the goal. There are roughly two parts to each policy; the first part attempts to reach a state where a_{switch} could be effective at switching to a different subpart, and the second part executes a_{switch} once. The optimality requirement is with respect to reward gathered along the way.

As with the subgoal-options module, the probabilistic expected time transition, state transition, and reward functions (*pett*, *pest*, and *per*, respectively) are defined for each $a_{\text{switch}} \in A_{\text{switch}}$. Similarly, let $u : S_{\text{bdry}} \longrightarrow \prod_{S \in \overline{S^j}} S$ be a mapping that unpacks the state in I^{j+1} into S_{split} and the appropriate subpart's state variables in I^j, filling in the rest of $\overline{S^j}$ randomly.

The join-on-state module maps $I^j \to I^{j+1}$ as follows:

- $\tau^{j+1}(\sigma^{j+1}, \alpha, n) = pett(u(\sigma^{j+1}), \alpha, n)$
- $t^{j+1}_{\text{bdry}}(\sigma^{j+1}, \alpha, n, \sigma'^{j+1}) =$
  $pest_{\text{bdry}}(u(\sigma^{j+1}), \alpha, n, u(\sigma'^{j+1}))$
- $r^{j+1}(\sigma^{j+1}, \alpha) = per(u(\sigma^{j+1}), \alpha)$

Intervening modules The point of breaking the macro-action framework into two modules is that there can be other modules in between the division into subparts and the creation of the boundary state semi-MDP. Intervening modules can manipulate, abstract, approximate, and solve different subparts separately, before they get merged back together.

Each intervening module should be careful only to modify state and action variables corresponding to a single subpart. They should never modify values of the state variable S_{split} or the action variable A_{switch}, and they should only modify the transition/reward dynamics associated with these in order to reflect the changes that happened in some subpart's state space. For instance, consider the case of using a subgoal-options module M^j to abstract a particular level l of the `nethack` domain, in between the split-on-state and join-on-state modules. The dynamics of the action values in A^j_{switch} that switch into and out of level l refer to x-y coordinates, but the dynamics of comparable A^{j+1}_{switch} action values need to refer to *locations*.

5.1.5 Reward-shaping module

In the field of reinforcement learning, the distance that an agent needs to travel between successive positive rewards is often great enough that forward search for reward fails or that assigning credit to certain actions for achieving the reward is very difficult. To alleviate this problem, the concept of *reward-shaping* suggests introducing a fake reward, a pseudo-reward, that can act like cookie crumbs along the way to the agent's actual reward. This pseudo-reward can be a lot easier to specify than other kinds of advice because it suggests to the agent what it should try to achieve but does not need to say what specific actions to take to achieve it. Ng *et al.* [54] give conditions under which a reward shaping function is guaranteed not to change the optimal policy.

The module hierarchy framework ostensibly performs planning, not reinforcement learning, but even so, the same sort of reward-shaping process can be useful to assist various modules in planning certain sub-tasks or in deciding how to parameterize themselves (see section 5.2).

The single input to a reward-shaping module M^j is a reward-shaping function $r_{\mathsf{shape}} : \overline{S^j} \times \overline{A^j} \longrightarrow \mathbb{R}$. This function may satisfy the potential function criterion of Ng *et al.*, in which case guarantees can be made about the loss incurred by this module, but the function may also be some arbitrary function.

There are a few different reward-shaping functions that would be of assistance in the gridworld example. One could give a small negative reward for decreasing the battery charge and a small positive reward for increasing it; this would encourage the robot to make sure that it doesn't run out of battery charge and would help compensate for the times during which the battery state variable is ignored. Another reward-shaping function could give a small positive pseudo-reward for picking up a package and a small negative pseudo-reward for dropping the package off; this would give the robot a subgoal of picking up a package. Both of these reward-shaping functions satisfy the potential function criterion in [54] and therefore do not change the optimal policy.

The abstract state and action variable sets that this module creates for I^{j+1} are:

- $\overline{S^{j+1}} = \overline{S^j}$

- $\overline{A^{j+1}} = \overline{A^j}$
- $\tau^{j+1}(\sigma, \alpha, n) = \tau^j(\sigma, \alpha, n)$
- $t_k^{j+1}(\sigma, \alpha, n, \sigma') = t_k^j(\sigma, \alpha, n, \sigma')$
- $r^{j+1}(\sigma, \alpha) = r^j(\sigma, \alpha) + r_{\text{shape}}(\sigma, \alpha)$

This slightly abuses notation by treating a state σ and an action α as in both I^{j+1} and in I^j, but this is fine because they have the same state and action spaces.

This reward-shaping module can be viewed in the more general context of giving advice to a system, i.e., not giving it a solution, but pointing out something that is likely to be advantageous. Actually, the whole of the module hierarchy currently is built on a lot of advice: the choice and placement of abstraction modules and module parameters are all forms of advice. The next section discusses how to lessen the dependence on human advice.

5.2 Modules attempting auto-parameterization

The above modules require user-entered parameters in order to be customized for abstracting a specific domain. It would be useful to have every module automatically set its own parameters, and indeed some abstraction types already do this and can be thought of as being parameterless. For others, however, even if they cannot function completely without human input, it would be nice to reduce the number of parameters to the fewest possible.

A module can customize itself to the domain at hand in two ways: it can work from the MDP model that it is given, or it can use the execution paths gathered from trial runs. Both kinds of data are used by existing abstraction methods, but it is easier for the former type to integrate into the module hierarchy. A module is guaranteed that it will be given an i-model to abstract, but while it may be simple for it to gain trial run data by recording the states visited and actions chosen in that i-model, it is not possible for the module to control the overall behavior of the system so as to explore specific areas of the i-model. Theoretically, the modules could be given turns at controlling exploratory behavior, e.g., from the bottom up so that lower abstract actions would stabilize before being used by upper modules. Even so, it seems that the frequent dynamic changes of representation would hamper efforts to learn parameters this way. The modules in this section therefore solely use the given MDP model.

5.2.1 Auto-subgoal-options module

This module is an extension of the subgoal-options module that automatically determines what the subgoals should be. It takes a set of state variables to find goals in, and it creates goals wherever the transition or reward dynamics are "interesting."

The inputs to an auto-subgoal-options module M^j are

- $\overline{S_{\text{subsume}}} \subseteq \overline{S^j}$, a set of state variables to create goals in and then subsume; and
- $\{a_{\text{goal}}\} \subseteq A^*$, a set of action values drawn from an action variable $A^* \in \overline{A^j}$ of I^j; this is the same set of action values as is given as input to a subgoal-options module.

The cross product of the state variables in $\overline{S_{\text{subsume}}}$ gives a smaller state space to find subgoals in. Each state σ in this cross product can be thought of as corresponding to many states in the i-model's whole state space. When considering these many corresponding states, if the transition and reward dynamics are mostly the same but are different at a few, then σ is considered interesting and is selected as a subgoal. More formally, the algorithm to find subgoals is as follows:

1. Vary the transition distribution t_k over all transition distributions for non-subsumed state variables $\overline{S^j} \backslash \overline{S_{\text{subsume}}}$. Recall that $\overline{S_k}$ is the set of state variables that are relevant to the transition distribution.

2. Vary the action a over all non-subsumed action values $\left[\bigcup_{A \in \overline{A^j}} A\right] \setminus \{a_{\text{goal}}\}$.

3. Vary the partial state s_{sub} over the relevant subsumed state variables $\overline{S_{\text{subsume}}} \cap \overline{S_k}$.

4. For each combination of t_k, a, and s_{sub},

   (a) Vary the partial state s_{nonsub} over the relevant non-subsumed state variables.

   (b) Notice how the conditional probability distribution over post-states, given by $cpd(n, s') = t_k(s_{\text{sub}} \cup s_{\text{nonsub}}, a, n, s')$, varies as s_{nonsub} is changed.

   (c) If there is more than one *cpd* as s_{nonsub} is varied, and if one is more than twice as frequent as all the rest, then add all non-zero state outcomes in the infrequent *cpd*s as goals.

The algorithm to find interesting subgoals based on reward is analogous to this algorithm that is based on transition dynamics.

As an example from the `nethack` domain, this module would note that, for most x-y coordinates, when a player chooses the *up* action the *level* state variable doesn't change. When the player happens to be standing at an up staircase, however, the level changes. This indicates that the stairway's x-y coordinates should be thought of as an interesting subgoal.

This process seems as though it could potentially iterate over many transition functions, partial states, and action values, and indeed it would if the representation of the t_k were flat. However, since the t_k are given by ADDs, this algorithm takes time proportional to the size of the ADD. In particular, the execution time is dependent on the number of paths from root to leaves in the t_k. This dependency on the ADD size rather than the state and action space sizes produces substantial savings in execution time.

5.2.2 Auto-state-aggregation module

The state aggregation module discussed in section 5.1.2 takes, as input, a state variable to operate on and a set of clusters for that state variable. Whether these parameters will work well for the domain at hand is clearly dependent both on the domain and on the parameters selected.

It is difficult to tell *a priori* what parameters are suited to a domain, i.e., what parameters will not cause the eventual solution to be unacceptably sub-optimal. It turns out that this is a difficult problem that can only be solved in general by determining the optimal value function for the domain before aggregating states.

Consider the gridworld example from section 4.1, and suppose that the goal is to aggregate states by battery level, as described above. If a set of clusters is not pre-specified, it would be good to find clusters automatically and ensure that they do not unreasonably decrease the expected reward.

Suppose that the auto-state-aggregation module pursues some policy for choosing states to aggregate into clusters, and suppose that the goal is to aggregate as much as possible while only having the expected reward decrease by less than a certain fraction. It is difficult to figure out this point of best aggregation precisely, but perhaps aggregating slightly too coarsely or slightly too finely still would give approximately the same expected reward. This would give the module a margin of error in figuring out which states to aggregate.

Whether such a margin of error exists was tested over several experiments. In each experiment, a certain policy was chosen to cluster states according to battery level, and the number of clusters was varied from the number of original states (i.e,. no clustering) to one (i.e., everything in the same cluster). The four clustering methods used are:

- *TopCluster*: a number of battery levels from the maximum down are clustered together, and all other battery levels are in their own cluster.

- *BottomCluster*: a number of battery levels from zero up are clustered together, and all other battery levels are in their own cluster.

- *EvenCluster*: battery levels are divided into clusters of equal size (within one, due to discretization).

- *RandomCluster*: battery levels are clustered together randomly to make the desired number of clusters.

These clustering methods were used to provide parameters to the state-aggregation module. A module hierarchy using this state-aggregation module and a policy-iteration module was used to solve the domain.

Figure 5-1 shows the results of the experiments. Unfortunately, it does not seem that expected reward slowly decreases as clustering is increased. Rather, expected reward remains relatively constant until the aggregation crosses a threshold, becomes too much, and pushes the reward down sharply and severely.

The intuition for these aggregation vs. reward curves has to do with aggregation violating the Markovian nature of the domain. The problematic part of the domain is where the robot's battery is running low and needs to make a beeline to the charger. Recall that the state-aggregation module creates cluster abstract dynamics that are the arithmetic average of the dynamics for states in that cluster. Suppose that some cluster (abstract state) C contains mostly states where the robot can reach the charger in time to charge but also contains one where it can't. If the robot reaches abstract state C during execution, then it may reason that it doesn't need to go to the charger yet because running too low on battery is an unlikely event. This decision is catastrophic, though, if the robot is on the edge of charger reachability given its current battery level. This catastrophic decision may be made because the abstract stochastic dynamics don't really reflect stochastic dynamics in the domain; instead, they are a substitute for attentiveness to precisely what the current concrete states are. The abstract domain has become non-Markovian, and that is what causes the

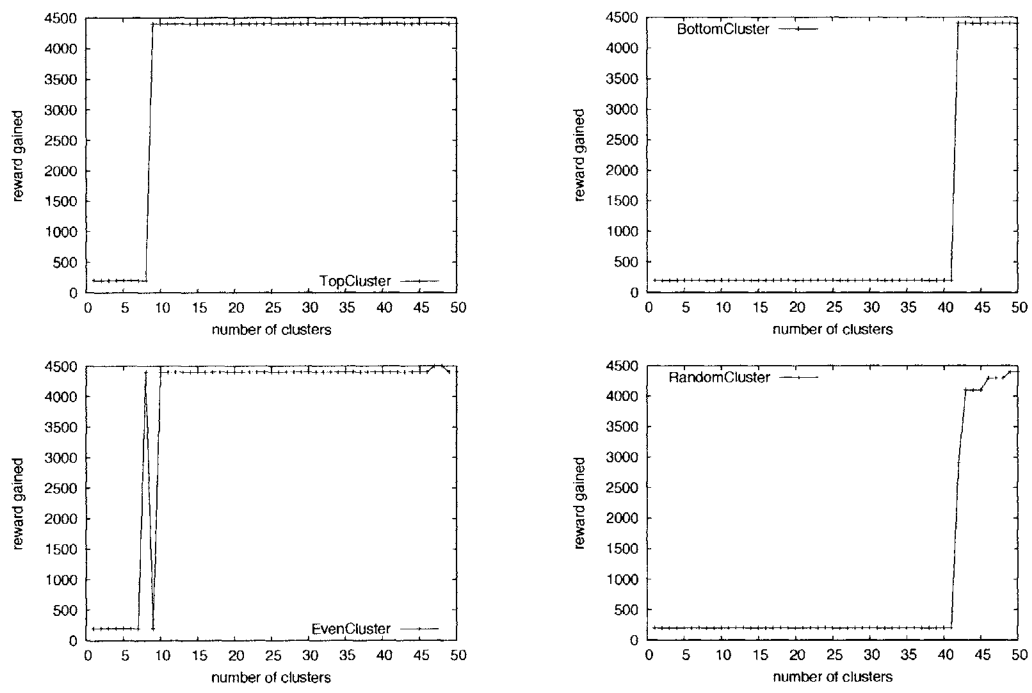


Figure 5-1: Experimental results for automatic parameterization of the state-aggregation module.

sharp decrease in expected reward.

Some prior methods have been used to try to automatically parameterize state aggregation through the use of heuristics [21], but these generally assume that the aggregated (abstract) domain is then solved directly and therefore a value function is being found for it, or they assume that some RL method like Q-learning can be done efficiently in the aggregated domain. These prior methods do not seem like they can be extended to operating in the middle of an abstraction hierarchy. Creating heuristics that function well in the middle of a module hierarchy remains an interesting area of future research.

5.2.3 ADD-reordering module

Having the transition probability distribution and reward functions expressed as algebraic decision diagrams (ADDs) is highly advantageous to the module hierarchy. One main advantage is that these transition and reward functions have a representation size that is proportional to the "complexity" of the function being represented (e.g., how uniform is it, does it have many special cases, etc.). Using a simple table would result in representation sizes proportional instead to the state and action space sizes.

The other main advantage is that the ADD representation exposes more structure in the transition dynamics and reward function. Abstraction modules therefore can take advantage of this structure and can generally operate on a whole i-model in time and using space proportional to the complexity of the i-model rather than in proportion to the size of the state and action spaces. Each one of the modules described here expects to operate on ADDs and uses the exposed structure in doing its abstraction.

Though ADDs offer a more compact representation than tables, decision trees, and other simple data structures, the smaller representation comes at the cost of needing to maintain the ADD invariant; recall that no subpart may be repeated in an ADD (identical subparts are merged). The penalty for merging repeated sub-diagrams is

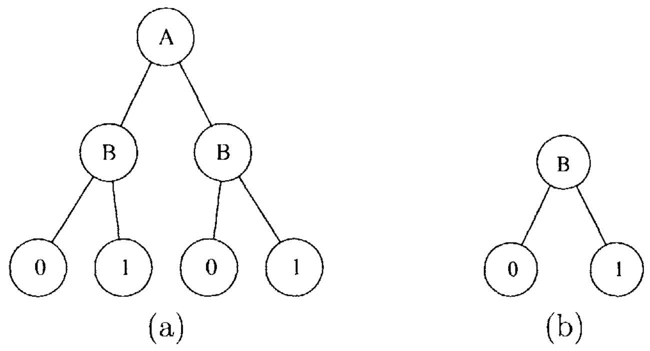


Figure 5-2: (a) An ADD in the middle of construction. (b) The completed ADD.

non-zero but is far outweighed by its benefits. It is precisely in maintaining this invariant that independences are automatically discovered. For instance, suppose an abstraction module is constructing a transition function ADD and builds the (non-reduced) decision diagram in figure 5-2 (a). When converted to an ADD, this decision diagram becomes the diagram in figure 5-2 (b), and all decision nodes for the state variable B have disappeared.

While ADDs generally represent functions in a compact way, the exact representation size is highly dependent on the ADD's *choice ordering*, the order of choices from the root to the leaves of the ADD. Varying an ADD's choice ordering may cause its size to vary by several orders of magnitude. It is therefore important that an appropriate choice ordering be used when representing an MDP's transition and reward functions as ADDs. Selecting a good choice ordering means that the representation will be smaller and the abstraction modules in the module hierarchy will have less work to do.

There will not necessarily be one choice ordering that is good for all i-models in the module hierarchy simultaneously, because the domain structure may change considerably as the domain is successively abstracted. It will therefore be advantageous to reorder the ADD choices in the middle of the hierarchy. This capability fits well into an ADD-reordering module.

There are two decisions to make about ADD choice reordering.

What new choice ordering to select The first and biggest question is how to reorder, i.e., what algorithm to use to determine the best choice ordering. Determining the optimal ordering is intractable [74], and so various heuristics have been developed to find an approximately optimal (or just a better) choice ordering.

It is necessary to select the objective function to be minimized. All ADD representations of the same function have the same number of terminals, so it really comes down to minimizing either the number of nodes in the ADD or the number of paths from root to leaves in the ADD. The abstraction work done by modules seems to be more proportional to the number of paths in an ADD than the number of nodes, so the number of paths is what the ADD-reordering abstraction module will attempt to minimize. There are several ADDs used in representing a domain (one to represent the reward function and several to represent the transition probability functions), and all or some subset of these could have the number of ADD paths jointly minimized. In running experiments, it turns out that the reward function ADD is a good proxy for the whole family of ADDs, that is, minimizing the reward function ADD's number of paths will generally cause the transition probability function ADDs' number of paths to be close to minimal as well. It is obviously advantageous to operate on just the reward function ADD, since it is less expensive to reorder the choices in one ADD than in many.

The simplest reordering strategy is to create several random choice orderings and take the best one. While this often ends up with some reduction in size, it gets nowhere near the best choice ordering found by Rudell sifting [73]. In Rudell sifting, each choice is in turn tried at every level from root to leaves (while the remaining choices stay in a fixed order with respect to each other). After a choice has been tested at all locations, it is moved to the location that minimized the objective function, and then the next choice is tested at all locations, and so on.

Random ordering and Rudell sifting are examples of general ADD reordering heuristics. The reordering occurs in a known context, however, so it may be that the module or modules directly below an ADD-reordering module may give clues as to a good ordering to use. Suppose that the module hierarchy contains a subgoal-options module between two ADD-reordering modules. The lower ADD-reordering module will select a good choice ordering. The subgoal-options module will then operate on the domain, adding and removing state and action variables, and changing the transition and reward dynamics accordingly. The upper ADD-reordering module may be able to use information about the state and action variables worked on by the subgoal-options module to make a good guess at a good new choice ordering. For instance, it may be that the reward ADD is more likely to be compact if the state and action variables that have values changed or that are added anew are at the root of the ADD rather than at the leaves (or vice versa).

These general and hinted reordering heuristics—random reordering, Rudell sifting, and bipartite Rudell sifting—were all added to an ADD-reordering module and tested against each other (see section 6.3).

When to select a new choice ordering The second decision to make about ADD choice reordering is where to place ADD-reordering modules in a module hierarchy. They could be sandwiched between every single other module, but that would likely lead to a lot of unnecessary reordering for little gain. Ideally, the ADDs involved in representing the domain would be reordered whenever they are significantly larger than (say, over twice as large as) they could be with an alternate choice ordering.

This is a difficult metric to use, since no good estimate exists for the optimal size of an ADD, and calculating the optimal size (or an approximation thereof) is just as difficult as calculating the optimal ADD itself (or an approximation thereof). It is relatively sure, though, that certain abstraction modules will likely have a large impact on the advantageousness of choice orderings, and certain abstraction modules will not. In the former category will be modules like subgoal-options, and in the latter will be modules like reward-shaping and state-aggregation.

Thus, ADD-reordering modules can be placed in the module hierarchy at levels where the current choice ordering, inherited up the hierarchy, is likely to be sub-optimal enough that the effort spent in finding a new ordering pays off further up the hierarchy.

5.2.4 Policy-iteration module

As described in section 4.4.1, the topmost module in any module hierarchy is a module that creates a complete policy for its concrete i-model. This module might use an abstraction method that solves the domain in some approximate way, like the envelope or MAXQ methods, or it might just use policy iteration. The latter is described here.

In order to fit the standard interface for abstraction modules and yet solve the entire domain, this module creates the following abstract i-model:

- $\overline{S^{j+1}} = \{onestate\}$, where $onestate = \{here\}$ is a state variable with a single value.
- $\overline{A^{j+1}} = \{oneaction\}$, where $oneaction = \{act\}$ is an action variable with a single value.
- $\tau^{j+1}(here, act, n) = \begin{cases} 1 & n = 1 \\ 0 & \text{otherwise} \end{cases}$.
- $t_k^{j+1}(here, act, 1, here) = 1$.
- $r^{j+1}(here, act) = 1$.

The transition and reward dynamics of this abstract i-model are obviously contrived, but that is not a problem. It would be possible to estimate the long-term expected reward gained by acting in the domain, and then set the expected reward of taking the single action *act* accordingly. There is no need, though, to go through the extra calculation steps, since that information could never change the system's actions.

This abstraction module performs modified policy iteration as described by Puterman and Shin [67].

Chapter 6

Experimental results

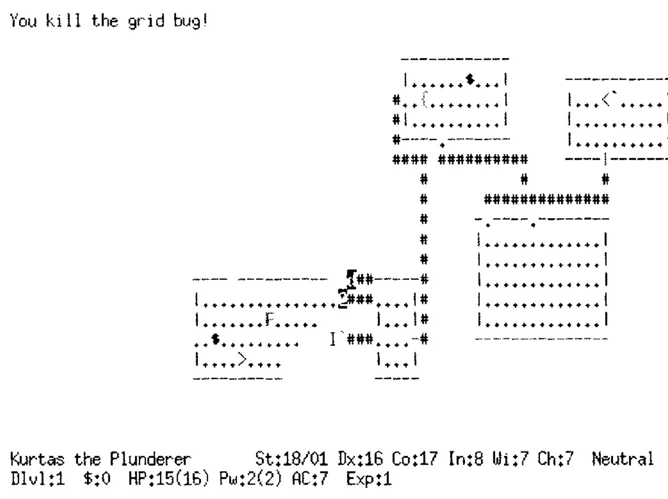
6.1 Example domain: `nethack`

In order to test the module hierarchy and many possible abstraction module variations, a simplified version of the computer game `nethack` was used (available on the world wide web at `http://www.nethack.org`; see figure 6-1). `Nethack` is a good domain for testing different approaches to solving real-world problems because it contains several different types of structure, some simple and some complex. The varying structure and the interaction between the different parts is representative of even larger, real-world domains, such a disaster relief robot, a Mars rover, or a general-purpose battlefield robot.

The `nethack` domain is a very good test domain for the following reasons:

- The domain has a simple core, consisting of things like movement, hunger, and health, but there is a large amount of depth and intricacy in a lot of areas; e.g., there are weapons whose effects are dependent on your player's class and race, your current skill level, the monster type, favor with your deity, and the current phase of the moon.

- The domain has stochastic action effects, stochastic observations, partial observ-



```
You kill the grid bug!

                                           ------------
                                           |......$...|      ------------
                                          #..{........|      |...<`.....|
                                          #|..........|      |..........|
                                          #----.-------      |..........|
                                          #### ##########    ---|--------
                                              #         #       #
                                              #     #############
                                              #    -.----.-------
                                              #    |............|
                                              #    |............|
                       ---- --------- 1##-----#    |............|
                       |.............2###....|#    |............|
                       |.......F.....    |...|#    |............|
                       ..$......... I`###....-#    --------------
                       |....>....        |...|
                       ----------        -----

Kurtas the Plunderer      St:18/01 Dx:16 Co:17 In:8 Wi:7 Ch:7  Neutral
Dlvl:1  $:0  HP:15(16) Pw:2(2) AC:7  Exp:1
```

Figure 6-1: A screenshot from the first level of the computer game `nethack`.

ability, sequential decisions, and reward (dying or winning). It fits the MDP framework ideally, taking advantage of the features that differentiate stochastic models from classical planning problems.

- The domain is so large that normal techniques don't even begin to approach solving it.

- The domain is not a large unstructured mess of states and actions, where there's nothing better to do than the baseline methods like policy iteration. The domain is also not so highly structured that a single specialized solver is the way to go (as with something like a stochastic traveling salesman problem). The `nethack` domain strikes a balance between these two extremes, having many aspects to the domain each with its own type of structure.

- The domain only allows one action at a time, so there is only one action variable, and it is not necessary yet to deal with the issues associated with concurrent actions.

Similar computer game domains `Nethack` is one computer game used as a test for MDP solvers designed for very large domains. Another such game is Warcraft, or its freely available version, Freecraft; the latter has been used in the work of Guestrin in his work on plan generalization in relational MDPs [26]. (Freecraft is now called Stratagus due to trademark issues with the owners of Warcraft and is available on the world wide web at `http://stratagus.sourceforge.net/`.) `Nethack` is similar to Freecraft in that there are overall goals to achieve, uncertainty in the world, and long-term consequences to actions. The domains are somewhat different because Freecraft has multiple agents to control, and each agent exists for a relatively short period of time; in `nethack`, on the other hand, there is only one agent to control, but that agent exists for the entirety of the game. `Nethack` is also much less forgiving than Freecraft: it is not uncommon for humans to be unable to advance beyond the first 20% or so of the game no matter how many times they try. The work of Guestrin *et al.* on RMDP plan generalization is mostly orthogonal to the work in this thesis on the module hierarchy and would be interesting to integrate into it in the future. See section 7.2 for more details on future plans to allow and exploit other types of structure.

Another computer game used for AI research has been Rogue (available on the world wide web at `http://users.tkk.fi/~eye/roguelike/rogue.html`). Rogue is a precursor to `nethack` and lacks its depth of gameplay but still retains many of its essential features: the goal is to explore the dungeon and retrieve artifacts, there are monsters to kill, weapons to wield, armor to wear, and so forth. Mauldin *et al.* created a system called Rog-o-matic [46] that was rather successful, winning the game and generally outperforming human experts. Rog-o-matic used hand-coded heuristics and an expert system architecture to play an early version of Rogue.

Simplified `nethack` In the simplified completely-observable version of `nethack` that was used for the experiments below, the goal is to escape from a dungeon.

```
WWWWWW  WWWWWW  WWWWWW
W<...W  W>...W  W....W
W....W  W....W  W....W
W....W  WWWW.W  WWWW.W
W....W      #       #
W..F.W  WWWW.W  WWWW.W
W....W  W...MW  W....W
W....W  W.<..W  W...<W
W...>W  W....W  W....W
WWWWWW  WWWWWW  WWWWWW
```

Figure 6-2: A small example dungeon with three levels; the highest level is on the left, and the lowest level is on the right. < denotes an up staircase, > denotes a down staircase, `F` denotes food, and `M` denotes a monster. The player starts in the top-left corner of the lowest level.

The dungeon is composed of several levels, where each level consists of some large rooms connected by narrow hallways. The levels are connected by stairways to the levels immediately above and immediately below them, and the escape stairway is at the top. (See figure 6-2 for a small example dungeon.) The player can move north, south, east, west, up, and down, but not diagonally.

The game is not just a path planning problem, because the player has hunger and health. The player starts out full but gets progressively more hungry as time goes on. If he starves, his health decreases, but there is food available to eat lying around the dungeon, and the player can carry this food with him. The player's health normally stays constant, but it decreases when starving or when attacked by a monster. The player can heal himself by using one of the medkits lying around the dungeon, and the player can pick up and carry medkits with him.

The domain was represented as an infinite-horizon discounted factored MDP with 11 primitive actions (*north*, *south*, *east*, *west*, *up*, *down*, *pickup*, *eat*, *heal*, *attack*, and *wait*) and a varying number of states depending on the exact layout of the dungeon. Some of the actions, such as movement and attacking a monster, had probabilistic outcomes (e.g., the monster dies with a high but not certain probability). The reward was set to be positive for escaping from the dungeon, negative for dying, and zero elsewhere.

6.1.1 Implemented module hierarchy

The module hierarchy that was created to solve the `nethack` domain is shown in figure 6-3. This module hierarchy uses eighteen modules instantiated from six module types. These modules were arranged in the module hierarchy and parameters were supplied by a domain expert, who tailored the structure and parameters so as to solve the simplified `nethack` domain as well as possible. It is important to note that, although the above module types were created in order to solve this simplified `nethack` domain, they are completely general and can be used in other module hierarchies to solve other problems, given appropriate parameter choices.

6.2 Comparison with individual methods

The running time and solution quality of the module hierarchy were compared both to policy iteration on the original domain and to several abstraction methods used in its modules, operating individually. Each method was run on a sequence of progressively more complicated instances of the `nethack` domain until it failed to escape from the dungeon within one hour. In successive domains, the number of items in the domain, the number of levels, and the x-y size of each level were gradually increased, so that the state space size ranged from 9,600 to 108,900,000. The number of primitive action steps required to escape from the dungeon increased sub-logarithmically with the size of the domain.

The running time results are given in figure 6-4. As expected, the module hierarchy is the only method that scales up to problems with very large numbers of states.

The most important point of comparison between the different methods is the size of the domains that each works with after having applied pertinent abstractions. The previous methods end up attempting to work in models with hundreds of thousands of states by the fourth or fifth test domain. In contrast, the largest model that the module hierarchy needs to solve has just 450 states and 15 actions. Granted, quite a

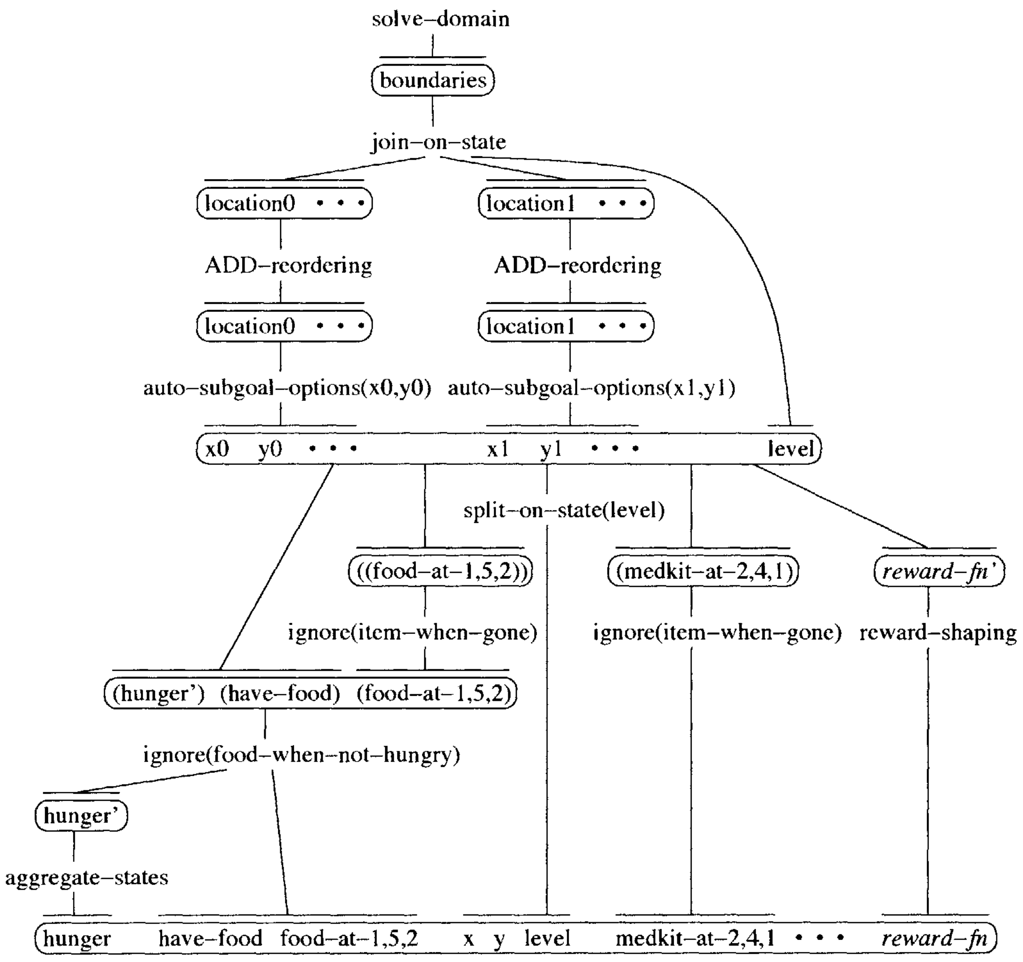


Figure 6-3: The module hierarchy used to solve the simplified `nethack` domain. Though the module hierarchy is a linear alternation of i-models and modules, modules are drawn showing which part of the domain they change, in order to better and more compactly illuminate the structure of the changes that each abstraction module makes.

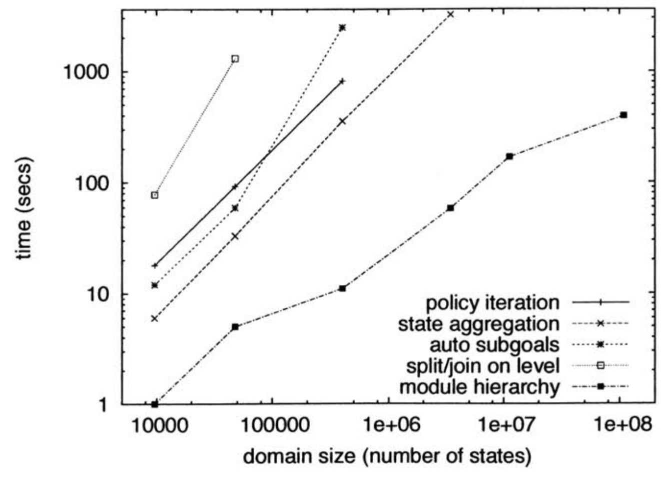


Figure 6-4: Experimental results, with execution times averaged over three runs each.

few 450-state domains are solved during the course of execution, but this is certainly preferable to intractability.

In the test runs described above, wherever two methods managed to produce a solution for the same test domain, the reward gained by those methods was the same (within the margin of error caused by the stochasticity of the domain and thus needing to average over several trial runs). In other words, all methods that succeeded in escaping from the dungeon in one CPU hour did so in approximately the same amount of time.

Of course, that no reward was lost by approximately solving the domains is due entirely to an appropriate choice of modules and their parameters. If, for instance, the selective-removal module that operates on the hunger portion of the domain were to have its threshold parameter set too low, then the player might die from hunger before being able to reach food. But an appropriate choice of modules is not unreasonable to assume, because any approximation method is dependent on the quality of its approximation. In addition, even if the approximation turns out to be wildly sub-optimal, a system that makes a very large domain tractable is still be better than one that can't handle the domain at all.

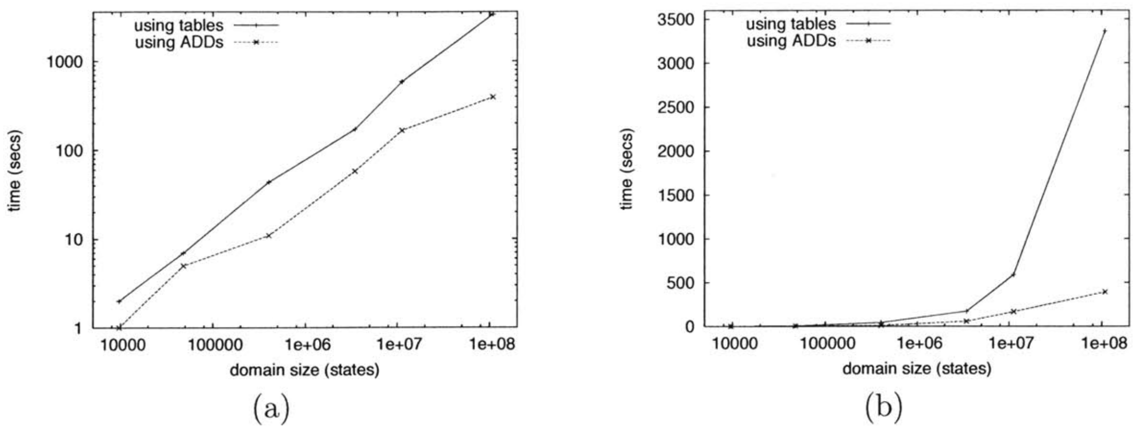


Figure 6-5: Experimental results for table- vs. ADD-based functions, with execution times averaged over three runs each, (a) as a log-log plot, and (b) as a semi-log plot.

6.3 ADDs: with and without and reordering

Using algebraic decision diagrams to represent functions in the MDP can be seen from figure 6-5 to be highly advantageous. Two sets of results are given for executing the same module hierarchy for the various `nethack` example domains from the previous section. In one set of results, the i-models store the transition and reward functions as tables; in the other set, the i-models store the functions as ADDs. Appropriate abstraction modules that can operate on the given data structure are used in each case, but they do not quite use the same algorithms internally, because a module operating on ADDs can use some shortcuts and exploit some structure that the same module operating on tables cannot.

The experimental results indicate that ADDs do not simply give a constant factor improvement over tables, but rather the benefit of using ADDs increases as the domain gets larger. In other words, the slightly flatter curve of the module hierarchy using ADDs indicates that ADDs help it to scale better.

As discussed in the section on the ADD-reordering module, different choice orderings can make large differences in the size of an ADD, and it is advantageous to optimize the choice ordering in the middle of a module hierarchy. Section 5.2.3

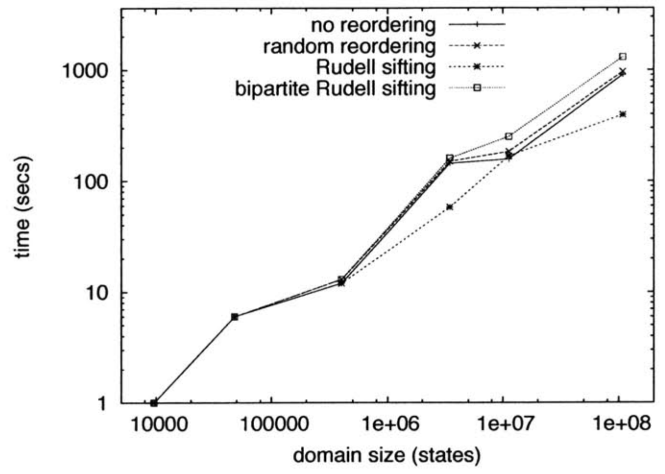


Figure 6-6: Experimental results for ADD choice reordering.

discusses several possible methods:

- Random reordering: testing n random choice orderings and choosing the best one (in the tests below, $n = 100$).
- Rudell sifting: testing each choice at all possible locations from root to leaves while holding the other choices fixed.
- Bipartite Rudell sifting: Rudell sifting where the choices corresponding to state and action variables changed by the abstraction module immediately down the hierarchy are constrained to lie only at the top (and, alternately, only at the bottom) of the choice ordering.

The results of the test are given below. Figure 6-6 shows the planning and execution time taken on the various `nethack` example domains when using each of these choice reordering methods, and when using no reordering at all.

Figure 6-6 reveals that Rudell sifting is the best among these algorithms. Random reordering does not manage to improve on the existing choice ordering, so it takes strictly longer than no reordering at all because of the time required to test the fruitless random choice orderings.

The perhaps surprising result is that Rudell sifting just barely improves upon doing no reordering whatsoever, and in fact it takes longer on the domain with around 10 million states. The reason that none of the reordering methods were able to improve the performance of the system by that much is that the benefit of dealing with smaller ADDs is offset by the work required to attain those smaller ADDs. Reordering ADD choices, at least as implemented here, is a process that takes a non-trivial amount of time; perhaps with further system tuning, its overhead could be reduced.

Chapter 7

Conclusions

The module hierarchy trades optimality and high speed on small domains for tractability on large domains. Of course, as usual when using this abstract representations, not only is optimality no longer guaranteed, but even being able to do "well" could be at risk. The reason that restrictions are not placed on how much expected reward a module is required to bound its loss by, though, is that it is undesirable to place *a priori* restrictions on the types of abstractions that can be used. Refusing a certain abstraction type because it could be used unwisely in theory means that it is unavailable to use wisely in practice. So, when solving very large domains, it is simply necessary to live with the possibility that an abstraction applied in an inappropriate way will of course cause lost opportunity for reward. This possibility allows the system to trade any reward necessary and be grossly suboptimal if that is required to achieve tractability.

Although the module hierarchy makes no guarantees about optimality, the `nethack` domain results show that it may not be necessary to sacrifice much optimality in order to gain tractability. Tractability is gained without losing optimality precisely because modules can be chosen to exploit the specific structure of different parts of the domain, and because those modules have the ability to re-abstract dynamically, changing the representation to focus domain solving on the (small) currently relevant portions of state space. It may be possible to give more precise estimates of the reward conceded given the specific types and parameters of actual abstractions used, but that is still future work at this point (see section 7.2).

7.1 Comparison with previous approaches

There are several previous approaches to hierarchical planning under uncertainty that are similar to module hierarchies, but most use a single abstraction method and exploit only one kind of domain structure. MAXQ [18], for instance, uses parameterized tasks arranged in a hierarchy to constrain the policy that it searches for, and it works very well on domains that decompose hierarchically into subtasks, but it does not support arbitrary other abstractions like state aggregation. Model minimization [12], on the other hand, performs state aggregation but no temporal abstraction.

The most similar previous work is the hierarchies of abstract machines (HAMs) framework [57] and its extension ALisp [1]. These methods allow for combining multiple abstraction types represented as non-deterministic finite state machines and Lisp programs, respectively. The module hierarchy framework has two advantages over these methods.

The first advantage is the representation of the domain MDP. The factoring of state and action spaces into variables, and the representation of functions as ADDs, allows for a large amount of structure to be stored explicitly with each i-model. This structure allows abstraction modules to operate on large chunks of state and action space at once; thus, a simple abstraction such as state approximation can operate on a domain in time proportional to the domain complexity rather than the domain size.

A lot of the benefit from the structured representation actually comes automatically from using ADDs to represent functions; for instance, ADDs never have redundant choices, so a function's independence of certain state variables manifests itself in the ADD structure with no special processing. Such independence is noticed and exploited in the split-on-state and join-on-state abstractors, for instance, when they determine that certain state variables are irrelevant to the workings of particular subparts and therefore do not need to be added to that subpart's state space.

The second and larger advantage of the module hierarchy comes in the ability to change the representation dynamically, at any time, and update the plan accordingly. This allows for partial plans to be created and then amended as needed. As long as the current representation is accurate enough to show the gist of far-future dynamics, the details can wait until the present situation has been dealt with and currently useful information is no longer relevant.

(Not planning the far future in detail has the added bonus that no work is wasted if the far future turns out differently than currently expected.)

As an example, in `nethack`, a player rarely has to worry about simultaneous imminent death from starvation and imminent death from monsters. By removing and then selectively adding back these aspects of the domain only when necessary, it is possible to deal with them separately. This effect is even more pronounced in larger, real-world domains, which have more areas of knowledge that are fairly specialized and thus do not interact much; such areas of knowledge produce a combinatorial explosion in the size of the state and action spaces unless dealt with separately.

Our usage of ADDs to represent probability distributions and reward functions is inspired by SPUDD [34] and APRICODD [78]. These algorithms attempt to improve value iteration by exploiting the domain structure exposed by ADDs, which is an orthogonal approach to that of the module hierarchy framework. Though the SPUDD and APRICODD algorithms could have been used when, say, creating options in the subgoal-options module, the sub-domains being solved were small enough that this would have brought no significant improvement.

7.2 Future work

The module hierarchy described in the sections above is a good first step towards being able to solve very large real-world stochastic problems. It can take the appropriate steps to use abstractions that fit well to the domain structure, and it can change the domain representation dynamically to help avoid considering unnecessary contingencies. Also, it incorporates features like a standard abstraction module interface and an MDP represented in a factored manner with ADD functions, and shows how that structure can be maintained and exploited.

There are many possible second and third and following steps that could be taken from this point. All have in mind the same eventual goal, the holy grail of a black-box agent that can be dropped into a situation and will act well in the domain without the need for human input, but they extend the module hierarchy in different interesting ways.

More structure While the factored semi-MDP model that the i-models use is a relatively compact way of representing a lot of domains, there are still a lot of relationships and independences and so forth that it cannot explicitly represent. Other researchers have devised additional ways of arranging the data in an MDP that does make these explicit.

These extended representations go by such names as independent choice logic [62], probabilistic STRIPS [17], first-order MDPs [9], relational MDPs [22], object-oriented Bayesian networks [44], and logical MDPs [40]. They share the common goal of thinking about the world in a first-order rather than propositional way. They attempt to separate the way that the world works (the possible types of object, the possible relationships between objects) from the actual instances of objects in some particular problem to be solved. Some of these models additionally express the MDP's dynamics and reward functions as logical expressions rather than as tables or ADDs or such.

Having a domain represented as some sort of logical relational MDP can be advantageous for several different reasons. One benefit is that the domain can often be expressed more compactly than even with factored semi-MDPs. Another is that planning or learning done in specific domain instances can often carry over to domains that are similarly structured but simply have different instances of objects (for examples, see the work of Guestrin *et al.* [28], of Kersting and De Raedt [40], and of Pasula *et al.* [58]).

Unfortunately, there seem to be no (or very few) abstraction methods that are designed specifically to exploit first-order and logical representations. If and when such methods are created, then it would be reasonable to change i-models to such a representation and be compatible with powerful first-order and logical abstraction methods.

Partially observable, continuous state space MDPs A big assumption in the MDP framework is that the current state is known at all times. This assumption does not hold for a lot of real-world domains; instead, the state space is partially observable, and adding partial observability dynamics to MDPs results in partially observable MDPs (POMDPs). POMDPs are generally much more difficult to solve optimally than MDPs of the same size. There has been some work on solving approximately solving large POMDPs [61, 72, 79, 80], but most of these methods are not abstraction methods (creating a new, simpler POMDP from the original POMDP) but instead solve the domain in some abstract way. There are a few POMDP abstraction methods, though, like the value-directed compression of Poupart and Boutilier [63] and Dean and Givan's extension of model minimization to POMDPs [23]. Were there to be more of these that were effective in simplifying POMDPs, then it would be advantageous to change the i-models to include partial observability dynamics.

One common way of dealing with discrete state- and action-space POMDPs is to turn them into continuous state- and action-space MDPs over belief space. The belief space of a POMDP is part of an n-dimensional hyperplane (n is the number of POMDP states) where all values lie between zero and one, and the sum of the coordinates for any point on the hyperplane is exactly one. A point in belief space represents an agent's belief about what probability it has of being in each state and is a sufficient statistic for all previous observations (thus ensuring the Markov property). It would thus be nice to allow continuous state and action spaces in the i-model representation, not only for domains that are continuous MDPs to begin with, but also to help solve MDPs using techniques that transform POMDPs into continuous state-space MDPs [72].

Optimizing re-planning One of the goals of the module hierarchy is to have different abstraction modules operate on precisely the kinds of structure they are good at abstracting, which will mean that most modules will abstract details away from only a small part of the domain's state and action spaces. Unfortunately, if some dynamic change happens near the bottom of the hierarchy, then every module from there up to the top needs to propagate the results of the change. The dynamic change may not have affected what a certain module did, but it needs to propagate the change's results nonetheless.

To assist a module in determining just how a dynamic change has affected its abstraction calculations, it would be useful to have some sort of structured way of describing how a dynamic change has caused an i-model to change. Given such a change language, abstraction modules could better understand what the dynamic change is all about and how its abstraction calculations are affected (or not affected, as the case may be).

As a further step towards diminishing the overhead related to re-planning caused by dynamic changes, it would be nice to have a way of telling when some dynamic change "doesn't matter," i.e., the abstract i-model that a module produces will remain the same or approximately the same despite certain modifications to the concrete i-model. This would involve some sort of sensitivity analysis, parsing the domain model to determine just how much effect a small perturbation to the transition dynamics has on the expected long-term reward. (Note that, while an arbitrarily small perturbation to transition dynamics can completely change the optimal policy, it will not change the expected long-term reward of a given policy by very much, giving this research direction some hope.)

Tighter optimality and goodness estimates In order to make headway on very large domains, the module hierarchy trades away optimality and even tight bounds on how approximately optimal solutions are. If such bounds are desired for some reason, it may be possible to derive them given that only certain types of modules and certain combinations thereof are used in building the module hierarchy.

One interesting approach would be to derive estimates for how well each module fits the structure that it is abstracting. The idea would be similar to the optimality estimates but is on the level of a single module, trying to figure out whether the module is retaining or throwing away important information, and therefore trying to figure out if the module is functioning effectively. For modules that re-abstract dynamically, and perhaps for others, these fit estimates will need to be derived at runtime. It is unclear whether such a fit estimation algorithm would be changed on a module-by-module basis, needing to know the inner details of each module, or if it can just look at the before and after intermediate models, or if it is even feasible at all. Also, it is unclear whether this fit estimate needs to be measured for each module individually (and then the estimates can be combined in some way), or whether it would need to operate on modules as they are arranged in a hierarchy.

Further automation and learning The module hierarchy may be a powerful tool to solve very large MDPs, but it requires quite a bit of guidance from a human domain expert to choose, arrange, and parameterize modules. A future goal is to lessen this dependence on human expertise.

The first step will be to determine automatically the parameters for the approximation and abstraction modules that currently rely on extra input from humans. This might be made easy given stable and accurate module fit estimates, since a module might not actually have to be inserted and run in order to tell if some parameters work well. It is likely that the parameter space for most modules would be large enough to require some sort of heuristic search. Perhaps the particular part of the domain on which the module operates, if small enough, would give clues as to good parameters.

The second and probably most difficult step will be to determine automatically not only the parameters for modules but also which module should go where, for any given domain. This would make the system almost totally autonomous, but this would be very tricky to achieve, since structure learning is known to be very difficult.

The third step would be to come up with new module types automatically, perhaps using the module fit estimator to do some rough search through approximation space. This may at first glance seem rather intractable, but if the module hierarchy scales up to thousands of modules, then it may be possible to solve a very large domain with lots and lots of very simple modules rather than a few complex ones.

It is unclear at this point whether there are actually that many simple modules to come up with; the techniques that humans normally use can probably be distilled to a few tens of modules (with appropriate parameters, of course). In perusing the space of previous abstraction methods, it is clear that they are all variations on a few themes (see section 2.3.1). In any case, the benefit of requiring only simple modules to operate is that the space of simple abstraction types is much smaller than the space of complex abstraction types. In addition, once a new module type is discovered and verified to be useful, it is no longer necessary to re-discover it, but it can be used as a part of the toolbox for all future module hierarchies.

A final step would be to learn the MDP itself, before or concurrently with building and executing a module hierarchy for it. It would actually be very interesting to try to combine MDP learning with the module hierarchy, because that might assist in the construction and parameterization of a module hierarchy. It is certainly easier to solve a smaller domain model, as an agent would have when just starting out in a domain, and the dynamic adaptive nature of the module hierarchy would pair well with a continuously changing model in the process of being learned. It is not unreasonable to model the insertion (or removal) of new modules in the hierarchy as dynamic change events, to be dealt with by propagating messages up the hierarchy.

Scaling to the real `nethack` Several of the steps above would be necessary in order to scale to the actual computer game `nethack` rather than a much simplified version. The most obvious step is that the module hierarchy would need to be able to deal with POMDPs as well as MDPs, since the `nethack` dungeon is very partially observable. The biggest challenge would not be figuring out i-model semantics or abstraction module interfaces but rather creating abstraction modules to function in POMDPs.

The other biggest challenge in scaling to the real `nethack` domain is exploiting all the structure that exists. This would involve creating a more structured representation, perhaps a logical one, to describe the dynamics of `nethack`. After all, the way that hunger increases and decreases and the way that armor and health interact are complicated but still very regular. These areas of the domain potentially can create transition function representations that are exponential in the number of features being described. Exploiting the structure would also involve creating the language for describing propagated changes to allow modules to be smarter in re-planning. Given the incredible size of and possibilities in the `nethack` universe, the vast majority will be irrelevant and ignorable at any time.

7.2.1 Contributions

This thesis makes several contributions to the field of artificial intelligence:

- This thesis explores using multiple qualitatively different abstraction methods together to solve the same domain. It shows how these abstraction methods can be packaged into modules and how those modules can operate together. While other methods like HAMs could theoretically provide the same functionality, this thesis describes an interface specifically designed to encapsulate abstraction methods so that they can be used as black boxes. It also gives convincing evidence that exploiting different types of structure in this way is necessary and useful.

- This thesis describes why and how to change the current abstract domain representation dynamically. It shows how to maintain a plan for the current representation and how to do so efficiently. It explains how a dynamic representation exploits the local nature of relevancy and provides evidence that this dynamism allows solving domains that would otherwise be intractable.

- This thesis describes a model for factored semi-MDPs that contains both temporally abstract actions and factored state and action variables. It discusses how this model is the most general that will fit within certain constraints and shows that the model is useful for maintaining the structure in information as it is successively abstracted and approximated.

- This thesis re-creates several existing abstraction methods in versions that take advantage of the structure of algebraic decision diagrams. Whereas before these methods would iterate over the entire state or action space in the process of creating abstract versions, they now need only iterate over the paths in the ADDs representing transition distribution functions. This makes their execution time dependent on the complexity of the domain rather than the size of the state space.

- This thesis implements the module hierarchy framework, combining the ideas of using multiple abstraction methods and of dynamic representation changes. It tests the module hierarchy framework implementation on non-trivial domains styled after the computer game `nethack` and demonstrates that the framework compares well to the non-multiple-method, non-dynamic alternatives.

7.3 Final thoughts

The module hierarchy described in this thesis is different than other methods of planning and acting in MDPs because it promises to scale to extremely large models. It can do so because its focus on being able to act tractably in the domain leads it to make tradeoffs comparable to humans: navigating only between landmarks, grouping similar situations together, following known sequences of actions, and so forth.

By following the plan outlined in the future work section above, it will be possible to improve the efficiency of the module hierarchy and to increase its scope of applicability to many interesting real-world domains. The resulting system promises to make artificial intelligence easier to deploy in real-world applications by having a library of components that can operate in a standard framework and are smart enough to arrange and configure themselves to solve any domain.

Appendix A

Programming details

The module hierarchy system described in this thesis was implemented and the implementation used to generate the various experimental results. The code consists of about 40,000 lines of code written in version 1.4 of the java programming language. The code was compiled and executed on an Apple PowerBook G4 running OS X 10.3.

The heart of the system consists of abstraction modules. Each abstraction module conforms to the interface given below, allowing it to participate in a complete module hierarchy.

```
public interface AbstractionModule {

  /**
   * <p>Returns the i-model directly above this abstraction module in
   * the module hierarchy, which this module keeps in sync with its
   * concrete i-model.</p>
   *
   * @return the abstract <code>Imodel</code>
   **/
  public Imodel getAbstractImodel();

  /**
   * <p>Returns the i-model directly below this abstraction module in
   * the module hierarchy, which this module monitors for changes and
   * uses to keep the abstract i-model in sync.</p>
   *
```

```
 * @return the concrete <code>Imodel</code>
 **/
public Imodel getConcreteImodel();

/**
 * <p>Ensures that the abstract i-model created by this abstraction
 * module is in sync with the concrete i-model.  An abstraction
 * module may be lazy about updating the abstract i-model when
 * notified that the concrete i-model has changed; when this method
 * is called, it means that the abstract i-model is about to be used
 * and must be up to date.  This method should always be called
 * right before processing the abstract i-model that an abstraction
 * module has created.</p>
 **/
public void ensureUpToDate();

/**
 * <p>Notifies this abstraction module that the concrete i-model has
 * changed.  This module should note that the change occurred and
 * then notify the abstraction module above it in the module
 * hierarchy that that higher module's concrete i-model (this
 * module's abstract i-model) has changed.  This change may be lazy;
 * the abstract i-model need not actually be updated until
 * <code>ensureUpToDate</code> is called.</p>
 **/
public void notifyConcreteImodelChanged();

/**
 * <p>Sets the atomic state (i.e., the state in the original, given
 * model) with respect to which this abstraction module should
 * create the current model dynamics.  This essentially means that
 * the given state should be representable in the abstract i-model,
 * i.e., the current state should not only be one that is reached in
 * the middle of a temporally-extended action.  When this method is
 * called, the abstraction module should make sure that modules
 * below in the module hierarchy are modeling dynamics for the given
 * state and re-create the abstract i-model as necessary.  The
 * method may be (and probably should be) lazy; the abstract i-model
 * need not actually be updated until <code>ensureUpToDate</code> is
 * called.</p>
 *
 * @param stateAtomic the atomic state needing to be representable
 **/
public void setAtomicStateToModel(State stateAtomic);

/**
 * <p>Converts a state in the concrete i-model into one in the
 * abstract i-model.  This method will only be called for concrete
 * states that have a representation in the abstract i-model;
 * currently, it is only called to convert the current state when
 * the abstract i-model is not in the middle of executing an action,
 * i.e., <code>this.isExecutingAction()</code> returns false.</p>
 *
 * @param stateConc a state in the concrete i-model
 * @return a state in the abstract i-model
 **/
public State getAbstractFromConcreteState(State stateConc);

/**
 * <p>Begins executing the given action in the abstract i-model.
 * This abstraction module is responsible for remembering the
 * currently executing action and determining when it has
 * terminated.</p>
 *
 * @param actionAbs an action in the abstract i-model
 **/
public void setAction(Action actionAbs);

/**
 * <p>Notifies this abstraction module of the current atomic state
 * (i.e., the current state in the original, given model).
 * Normally, a module will want to use the
 * <code>getAbstractFromConcreteState</code> methods of lower
 * modules to convert the current atomic state into a state in the
 * concrete i-model when needed.</p>
 **/
public void makeObservation(State stateAtomic);

/**
 * <p>Queries whether the action in the abstract i-model that was
 * last executed has finished executing yet.  A module should use
 * the last observed current state to decide whether the currently
 * executing action in the abstract i-model has finished.</p>
 *
```

```
   * @return whether the abstract action has finished
   **/
  public boolean isExecutingAction();

  /**
   * <p>Returns the next atomic action corresponding to the currently
   * executing action in the abstract i-model.  Generally, a model
   * will use the abstract action to choose actions in the concrete
   * i-model, will set those actions in the next module down the
   * hierarchy, and will ask that lower module for the next atomic
   * action.</p>
   *
   * @return the next atomic action to execute
   **/
  public Action getNextAtomicAction();
}
```

Bibliography

[1] D. Andre and S. Russell. State abstraction for programmable reinforcement learning. Technical Report UCB//CSD-02-1177, University of California at Berkeley, Computer Science Division, Berkeley, CA, 2002.

[2] J. Baum and A. Nicholson. Dynamic non-uniform abstractions for approximate planning in large structured stochastic domains. In H. Lee and H. Motoda, editors, *Proceedings of the Fifth Pacific Rim International Conference on Artificial Intelligence*, pages 587–598. Springer, 1998.

[3] R. Bellman. *Dynamic programming.* Princeton University Press, 1957.

[4] D.B. Bersekas and D.A. Castañon. Adaptive aggregation for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 34(6):589–598, 1989.

[5] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In S. Chien, S. Kambhampati, and C.A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 52–61. AAAI, 2000.

[6] C. Boutilier, R. Brafman, and C. Geib. Prioritized goal decomposition of Markov decision processes: towards a synthesis of classical and decision theoretic planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1156–1163. Morgan Kaufmann, 1997.

[7] C. Boutilier and R. Dearden. Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1016–1022. AAAI, 1994.

[8] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1104–1111. Morgan Kaufmann, 1995.

[9] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In B. Nebel, editor, *Proceedings of the Seventeenth International*

Joint Conference on Artificial Intelligence, pages 690–700. Morgan Kaufmann, 2001.

[10] D. Castañon. Approximate dynamic programming for sensor management. In *Proceedings of the Thirty-Sixth IEEE Conference on Decision and Control*, pages 1208–1213. IEEE, 1997.

[11] P. Dayan and G.E. Hinton. Feudal reinforcement learning. In S.J. Hanson, J.D. Cowan, and C.L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 271–278. Morgan Kaufmann, 1993.

[12] T. Dean and R. Givan. Model minimization in Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 106–111. AAAI/MIT, 1997.

[13] T. Dean, R. Givan, and S. Leach. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In D. Geiger and P.P. Shenoy, editors, *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 124–131. Morgan Kaufmann, 1997.

[14] T. Dean, L.P. Kaelbling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 574–579. AAAI/MIT, 1993.

[15] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[16] T. Dean and S. Lin. Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1995.

[17] R. Dearden and C. Boutilier. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(1–2):219–283, 1997.

[18] T. Dietterich. The MAXQ method for hierarchical reinforcement learning. In J.W. Shavlik, editor, *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126. Morgan Kaufmann, 1998.

[19] T. Dietterich and N. Flann. Explanation-based learning and reinforcement learning: a unified view. In A. Prieditis and S.J. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 176–184. Morgan Kaufmann, 1995.

[20] D. Dubois, M. Grabisch, F. Modave, and H. Prade. Relating decision under uncertainty and multicriteria decision making models. *International Journal of Intelligent Systems*, 15(10):967–979, 2000.

[21] Y. Engel and S. Mannor. Learning embedded maps of Markov processes. In C.E. Brodley and A.P. Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 138–145. Morgan Kaufmann, 2001.

[22] N. Gardiol and L.P. Kaelbling. Envelope-based planning in relational MDPs. In S. Thrun, L.K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT, 2004.

[23] R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1–2):162–223, 2003.

[24] R. Givan, S. Leach, and T. Dean. Bounded-parameter Markov decision processes. *Artificial Intelligence*, 122(1–2):71–109, 2000.

[25] C. Guestrin and G. Gordon. Distributed planning in hierarchical factored MDPs. In A. Darwiche and N. Friedman, editors, *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, pages 197–206. Morgan Kaufmann, 2002.

[26] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1003–1010. Morgan Kaufmann, 2003.

[27] C. Guestrin, D. Koller, and R. Parr. Multiagent planning with factored MDPs. In T.G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 1523–1530. MIT, 2001.

[28] C. Guestrin, D. Koller, and R. Parr. Solving factored POMDPs with linear value functions. In *Proceedings of the IJCAI workshop on planning under uncertainty and incomplete information*, 2001.

[29] C. Guestrin and D. Ormoneit. Robust combination of local controllers. In J.S. Breese and D. Koller, editors, *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 178–185. Morgan Kaufmann, 2001.

[30] M. Hauskrecht. Planning with macro-actions: effect of initial value function estimate on convergence rate of value iteration.

[31] M. Hauskrecht, N. Meuleau, L.P. Kaelbling, T. Dean, and C. Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In G.F. Cooper and S. Moral, editors, *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 220–229. Morgan Kaufmann, 1998.

[32] R. Heckendorn and C. Anderson. A multigrid form of value iteration applied to a Markov decision problem. Technical Report CS-98-113, Colorado State University, Department of Computer Science, Fort Collins, CO, 1998.

[33] B. Hengst. Discovering hierarchy in reinforcement learning with HEXQ. In C. Sammut and A.G. Hoffmann, editors, *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 243–250. Morgan Kaufmann, 2002.

[34] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: stochastic planning using decision diagrams. In K.B. Laskey and H. Prade, editors, *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288. Morgan Kaufmann, 1999.

[35] J. Hoey, R. St.Aubin, A. Hu, and C. Boutilier. Optimal and approximate stochastic planning using decision diagrams. Technical Report TR-2000-05, University of British Columbia, Department of Computer Science, Vancouver, BC, 2000.

[36] R. Howard. *Dynamic programming and Markov processes.* MIT, 1960.

[37] L.P. Kaelbling. Hierarchical learning in stochastic domains: preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 167–173. Morgan Kaufmann, 1993.

[38] L.P. Kaelbling. Learning to achieve goals. In R. Bajcsy, editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1094–1099. Morgan Kaufmann, 1993.

[39] M.J. Kearns, Y. Mansour, and A.Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1324–1231. Morgan Kaufmann, 1999.

[40] K. Kersting and L. De Raedt. Logical Markov decision programs. In *Proceedings of the IJCAI workshop on learning statistical models from relational data*, pages 63–70, 2003.

[41] M. Koivisto and K. Sood. Exact Bayesian structure discovery in Bayesian networks. *Journal of Artificial Intelligence Research*, 5(May):549–573, 2004.

[42] D. Koller and R. Parr. Computing factored value functions for policies in structured MDPs. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1332–1339. Morgan Kaufmann, 1999.

[43] D. Koller and R. Parr. Policy iteration for factored MDPs. In C. Boutilier and M. Goldszmidt, editors, *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 326–334. Morgan Kaufmann, 2000.

[44] D. Koller and A. Pfeffer. Object-oriented Bayesian networks. In D. Geiger and P.P. Shenoy, editors, *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 302–313. Morgan Kaufmann, 1997.

[45] T. Lane and L.P. Kaelbling. Nearly deterministic abstractions of Markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 260–266. AAAI, 2002.

[46] M. Mauldin, G. Jacobson, A. Appel, and L. Hamey. ROG-O-MATIC: a belligerent expert system. Technical Report CMU-CS-83-144, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1983.

[47] A. McCallum. *Reinforcement learning with selective perception and hidden state.* PhD thesis, University of Rochester, Department of Computer Science, Rochester, NY, December 1995.

[48] A. McGovern and A.G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In C.E. Brodley and A.P. Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 361–368. Morgan Kaufmann, 2001.

[49] A. McGovern and R.S. Sutton. Macro-actions in reinforcement learning: an empirical analysis. Technical Report 98–70, University of Massachusetts, Amherst, Department of Computer Science, Amherst, MA, 1998.

[50] A. McGovern, R.S. Sutton, and A.H. Fagg. Roles of macro-actions in accelerating reinforcement learning. In *Proceedings of the 1997 Grace Hopper Celebration of Women in Computing*, pages 13–18, 1997.

[51] N. Meuleau, M. Hauskrecht, K. Kim, L. Peshkin, L.P. Kaelbling, T. Dean, and C. Boutilier. Solving very large weakly coupled Markov decision processes. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 165–172. AAAI/MIT, 1998.

[52] A. Moore, L. Baird, and L.P. Kaelbling. Multi-value-functions: efficient automatic action hierarchies for multiple goal MDPs. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1316–1323. Morgan Kaufmann, 1999.

[53] R. Munos and A. Moore. Rates of convergence for variable resolution schemes in optimal control. In P. Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 647–654. Morgan Kaufmann, 2000.

[54] A. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: theory and application to reward shaping. In I. Bratko and S. Dzeroski, editors, *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann, 1999.

[55] A. Nicholson and L.P. Kaelbling. Toward approximate planning in very large stochastic domains. In *Proceedings of the AAAI Spring Symposium on Decision Theoretic Planning*, pages 190–196, 1994.

[56] R. Parr. Flexible decomposition algorithms for weakly coupled Markov decision problems. In G.F. Cooper and S. Moral, editors, *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 422–430. Morgan Kaufmann, 1998.

[57] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In M.I. Jordan, M.J. Kearns, and S.A. Solla, editors, *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT, 1998.

[58] H. Pasula, L. Zettlemoyer, and L.P. Kaelbling. Learning probabilistic relational planning rules. In S. Zilberstein, J. Koehler, and S. Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pages 73–82. AAAI, 2004.

[59] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann, 1988.

[60] J. Pineau, N. Roy, and S. Thrun. A hierarchical approach to POMDP planning and execution. In *Proceedings of the ICML workshop on hierarchy and memory in reinforcement learning*, 2001.

[61] J. Pineau and S. Thrun. An integrated approach to hierarchy and abstraction for POMDPs. Technical Report CMU-RI-TR-02-21, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 2001.

[62] D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):7–56, 1997.

[63] P. Poupart and C. Boutilier. Value-directed compression of POMDPs. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 1547–1554. MIT, 2003.

[64] D. Precup and R.S. Sutton. Multi-time models for temporally abstract planning. In M.I. Jordan, M.J. Kearns, and S.A. Solla, editors, *Advances in Neural Information Processing Systems 10*, pages 1050–1056. MIT, 1998.

[65] D. Precup, R.S. Sutton, and S. Singh. Theoretical results on reinforcement learning with temporally abstract options. In C. Nedellec and C. Rouveirol, editors, *Proceedings of the Tenth European Conference on Machine Learning*, pages 382–393. Springer, 1998.

[66] M. Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 1994.

[67] M. Puterman and M. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24(11):1127–1137, 1978.

[68] B. Ravindran and A.G. Barto. Symmetries and model minimization of Markov decision processes. Technical Report 01-43, University of Massachusetts, Amherst, Department of Computer Science, Amherst, MA, 2001.

[69] B. Ravindran and A.G. Barto. Model minimization in hierarchical reinforcement learning. In S. Koenig and R.C. Holte, editors, *Proceedings of the Fifth International Symposium on Abstraction, Reformulation and Approximation*, pages 196–211. Springer, 2002.

[70] B. Ravindran and A.G. Barto. SMDP homomorphisms: an algebraic approach to abstraction in semi-Markov decision processes. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1011–1018. Morgan Kaufmann, 2003.

[71] K. Rohanimanesh and S. Mahadevan. Decision-theoretic planning with concurrent temporally extended actions. In J.S. Breese and D. Koller, editors, *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 472–479. Morgan Kaufmann, 2001.

[72] N. Roy and G. Gordon. Exponential family PCA for belief compression in POMDPs. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 1635–1642. MIT, 2003.

[73] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 42–47, 1993.

[74] D. Sieling. The nonapproximability of OBDD minimization. *Information and Computation*, 172(2):103–138, January 2002.

[75] S. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[76] S. Singh and D. Cohn. How to dynamically merge Markov decision processes. In M.I. Jordan, M.J. Kearns, and S.A. Solla, editors, *Advances in Neural Information Processing Systems 10*, pages 1057–1063. MIT, 1998.

[77] D. Smith and D. Weld. Conformant Graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 889–896. AAAI/MIT, 1998.

[78] R. St-Aubin, J. Hoey, and C. Boutilier. APRICODD: approximate policy construction using decision diagrams. In T.K. Leen, T.G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 1089–1095. MIT, 2001.

[79] G. Theocharous and S. Mahadevan. Approximate planning with hierarchical partially observable Markov decision processes for robot navigation. In *IEEE International Conference on Robotics and Automation*, pages 1347–1352. IEEE, 2002.

[80] S. Thrun. Monte Carlo POMDPs. In S.A. Solla, T.K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1064–1070. MIT, 2000.

[81] W. Uther and M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 769–774. AAAI/MIT, 1998.

[82] W. Uther and M. Veloso. TTree: tree-based state generalization with temporally abstract actions. In S. Koenig and R.C. Holte, editors, *Proceedings of the Fifth International Symposium on Abstraction, Reformulation and Approximation*, pages 308–315. Springer, 2002.

[83] G. Wang and S. Mahadevan. A greedy divide-and-conquer approach to optimizing large manufacturing systems using reinforcement learning.

[84] G. Wang and S. Mahadevan. Hierarchical optimization of policy-coupled semi-Markov decision processes. In I. Bratko and S. Dzeroski, editors, *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 464–473. Morgan Kaufmann, 1999.

[85] K. Yost and A.R. Washburn. The LP/POMDP marriage: optimization with imperfect information. *Naval Research Logistics*, 47:607–619, 2000.